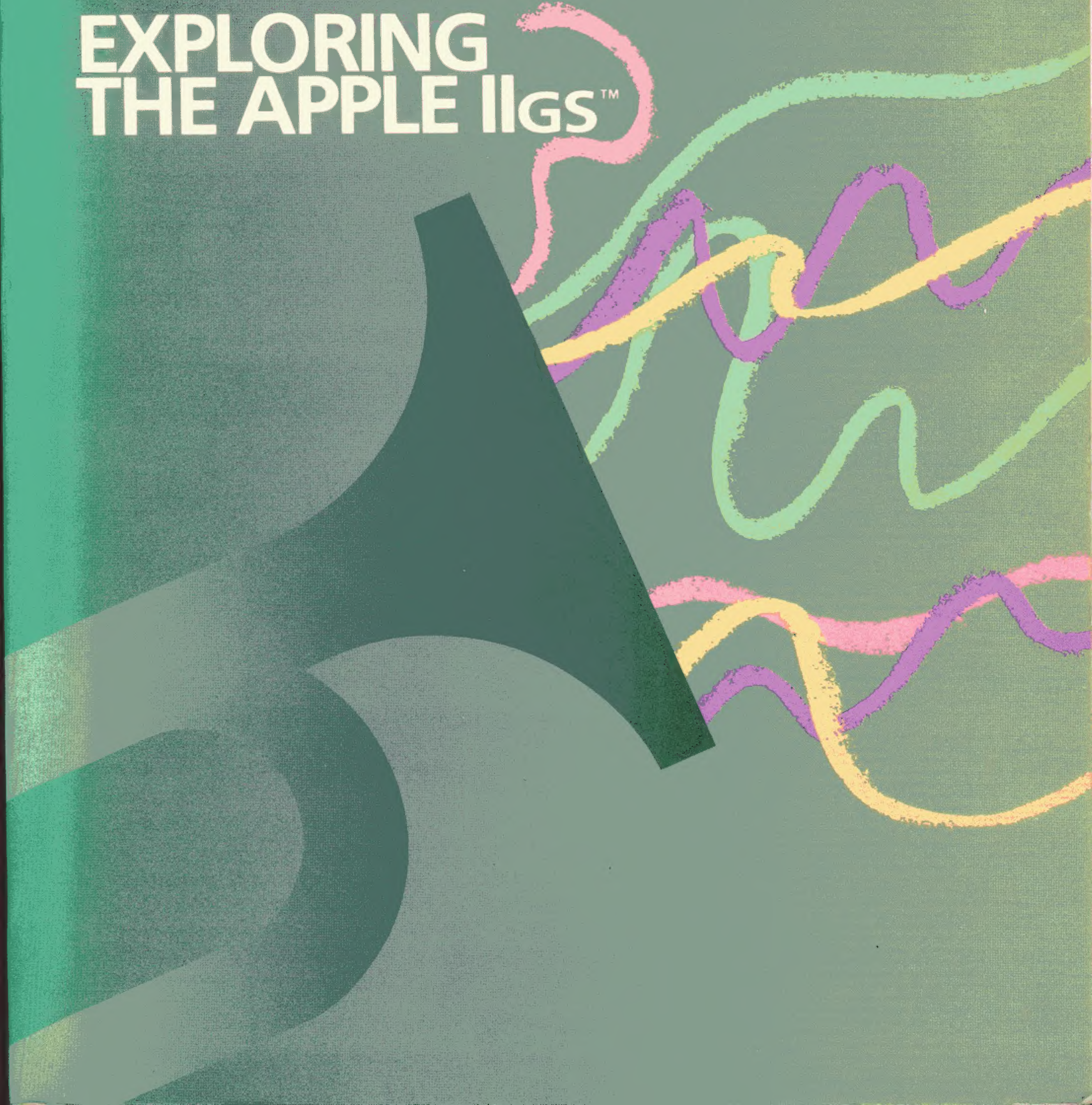
Addison-Wesley Publishing Company

GARY B. LITTLE

EXPLORING THE APPLE IIGS™

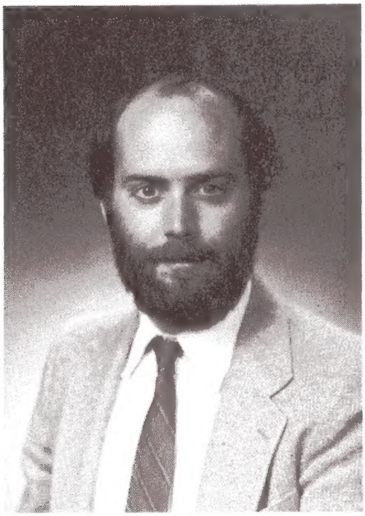


Exploring the Apple IIGS

This book is dedicated to my wonderful new daughter,

Adrienne Blair Little

About the Author



Gary B. Little lives in Vancouver, British Columbia. He is a founding member of Apple's British Columbia Computer Society and is active in several business organizations that promote and assist software developers. He is a contributing editor for A+ magazine and has written four other books on how to program Apple computers: *Inside the Apple IIe*, *Inside the Apple IIc*, *Apple ProDOS: Advanced Features for Programmers*, and *Mac Assembly Language: A Guide for Programmers*. Gary is also the developer of the Point-to-Point communications program and the Binary II file format.

Exploring the Apple IIGS

GARY B. LITTLE

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario ·
Wokingham, England · Amsterdam · Bonn · Sydney · Singapore
Tokyo · Madrid · Bogotá · Santiago · San Juan

Library of Congress Cataloging-in-Publication Data

Little, Gary B., 1954–
Exploring the Apple IIGS.

Bibliography.
Includes index.
1. Apple IIGS (Computer) I. Title. II. Title:
Exploring the Apple IIGS.
QA76.8.A66235L58 1987 004.165 87-11541
ISBN 0-201-15539-7

Cover design by Doliber Skeffington Design
Text design by Kenneth J. Wilson
Set in 10pt Caledonia by DEKR Corporation

ABCDEFGHIJ-HA-8987
First printing, August, 1987

PREFACE

The Apple IIGS™ computer is the most powerful member of the Apple® II family of microcomputers. Not surprisingly, it is also the most complex and the most difficult to program.

The purpose of this book is to show you how to develop software that takes advantage of the unique features of the GS without forcing you to sift through the thousands of pages of technical information available from Apple Computer, Inc. The major new features I will cover include the enhanced operating system (ProDOS® 16), a powerful sound system, two super high-resolution color graphics display modes, and hundreds of standard subroutines, called *functions*, a program can use to perform various operations.

The GS operating system groups related software functions into data structures called *tool sets*. Two examples of tool sets are the Memory Manager (which controls memory usage) and the Event Manager (which deals with user input activity). The GS also has several tool sets you can use to develop programs that take advantage of the desktop environment popularized by the Macintosh™ computer; the main ones are QuickDraw II (for drawing on the graphics screen), the Menu Manager (for implementing pull-down menus), the Window Manager (for handling multiple windows on the graphics screen), and the Dialog Manager (for handling user selections).

All the commonly-used tool sets are analyzed in this book. For convenience, tool set reference tables are included at the end of every chapter that reviews a tool set. For each function, these tables contain the function number and the list of parameters the function expects to receive when called.

The programming examples in this book were written in 65816 assembly language. (There is a good reason for this: high-level language compilers for C, Pascal, and BASIC were not available when this book was written.) As a result, you will find this book more valuable if you have assembly language experience. Even if you do not, however, the descriptions of what the tool set functions do and how they interact with one another will be useful. Converting assembly-langauge function calls to high-level langauge commands should not be difficult.

I give an overview of 65816 assembly language programming techniques in chapter 2. If you are a total stranger to assembly language programming, I suggest you first read a book totally devoted to the topic. The best are *65816/65802 Assembly Language Programming* by Michael Fischer and *Programming the 65816* by David Eyes and Ron Lichty.

As you will quickly discover while reading this book, the GS supports most of the hardware and software features of the Apple® IIe and IIc computers. These features have been exhaustively covered in three of my earlier books, *Inside the Apple IIe*, *Inside the Apple IIc*, and *Apple ProDOS: Advanced Features for Programmers*. The latter book describes what is now called the ProDOS® 8 operating system.

It would not have been possible to write a book like this so soon after the introduction of the GS without the technical support of Apple Computer, Inc. My thanks to all those at Apple who put up with my many telephone calls and electronic letters. Those who were particularly helpful were Rob Moore, Jim Merritt, and Steve Glass.

Gary B. Little
Vancouver, British Columbia, Canada
June 1987

CONTENTS

CHAPTER 1

Exploring the Apple IIGS

On September 15, 1986, Apple Computer, Inc. revealed the Apple IIGS™ computer, the first new model in the Apple® II family since the Apple® IIc (April, 1984) and the Apple® IIe (January, 1983) computers. The GS portion of the name stands for Graphics and Sound, the two exciting features of the GS that distinguish it from the IIe, IIc, and, in many ways, from most other microcomputers as well.

Building the GS was not an easy task for Apple's engineers because of one critical design constraint: the GS, even with its fancy new sound, graphics, and processing capabilities, had to be able to run software originally developed for the IIe and IIc. Nevertheless, the project was successfully completed. The result is a computer that runs almost all software available for the IIe or IIc but that has enough unique new features to keep programmers and users satisfied for years to come.

This chapter reviews the important hardware features of the GS while highlighting the differences between the GS and the IIe and IIc systems. Also provided is a cursory look at the new operating system software Apple provides with the GS. Once you are familiar with the overall makeup of the GS, you will quickly recognize its potential from a programmer's point of view. Subsequent chapters examine how to develop software for the GS that takes full advantage of its new features.

MICROPROCESSOR

The GS uses The Western Design Center W65C816 microprocessor, not the 65C02 used in the IIe and IIc. The 65816, as it is more commonly known, is a versatile device that can operate in an 8-bit emulation mode in which it mimics the operation of a 65C02 almost exactly, or in a more powerful 16-bit native mode. (See Appendix 2 for a technical description of the 65816.)

The 65816's ability to emulate the 65C02 made it the obvious choice as the microprocessor for the GS. Without this ability to emulate, the GS would not be able to run IIe and IIc software unless a slow software interpreter or (heaven forbid) the 65C02 were used.

Another important factor in choosing the 65816 was that it also has enough horsepower to handle rigorous new programming chores, such as managing large amounts of memory and performing complex color graphics operations. In its native operating mode, the 65816 supports the following features: 16-bit registers, more instructions, more addressing modes, and a relocatable direct page and stack. The 65816 also has a 24-bit address bus, so it can directly address a 16-megabyte memory space.

The 65816 operates at either 1.0 MHz (normal mode) or 2.8 MHz (fast mode) on the GS. The default mode is the fast mode, in which programs run about two and one-half times faster than they do at normal speed. The 65C02 on the IIe or IIc runs at only 1.0 MHz, although it is possible to purchase accelerator boards for the IIe to speed it up by a factor of three or more.

MEMORY

The address space for the 65816 is an enormous 16 megabytes and is directly addressable (that is, every byte has a unique address). Contrast this with the 65C02, which has only a 64K address space; additional memory duplicates these addresses and can be accessed only using relatively complex bank-switching techniques.

Chapter 4 takes a detailed look at how the GS makes use of the 16M address space. In brief, a standard GS has 256K of RAM, expandable to 8M, and 128K of ROM, expandable to 1M. (It also has RAM to support the sound system and the Control Panel, but this memory is not part of the 65816 address space.) To add extra memory, you must insert a memory expansion card in a special peripheral slot on the GS motherboard.

Of the 256K of RAM, the first 128K is roughly equivalent to main and auxiliary memory on the IIe and IIc. The other 128K is needed to support the operating system, system monitor, device drivers, and special video display modes.

The 128K ROM space on the GS contains a great many support programs and subroutines a programmer can use:

- The same Applesoft® programming language as in the IIe and IIc
- A System Monitor similar to the one on the IIe and IIc but with several new commands
- A Mini-Assembler like the one on the enhanced IIe and IIc, but extended to support all 65816 instructions and addressing modes
- System diagnostics
- Built-in classic desk accessory programs: Control Panel (for setting the system configuration) and Alternate Display Mode (for allowing IIe- and IIc-style programs to use page 2 of text)
- Drivers for built-in input/output (I/O) devices

- A driver for the AppleTalk® network

- Soft switches for configuring the system and performing I/O operations (these are actually memory-mapped I/O locations)

- Tool sets: Tool Locator, Memory Manager, Miscellaneous Tool Set, QuickDraw II, Sound Manager, Desk Manager, Floating-Point Numerics (SANE™), Apple DeskTop Bus™ Tool Set, Scheduler, Integer Math Tool Set, Text Tool Set, Event Manager, and RAM Disk Tool Set

A tool set is made up of several subroutines (or functions), each performing the same general type of operation. The Memory Manager tool set, for example, contains several functions for doing such chores as allocating blocks of memory, releasing blocks, calculating free space, and performing other memory-related operations. Most of this book deals with how to use tool set functions in your own programs.

I/O INTERFACES

From an I/O point of view, the GS combines the best features of the IIe with the best features of the IIc. Like the IIe, the GS contains seven general-purpose peripheral expansion slots numbered from 1 to 7. As a result, almost any peripheral board originally designed for the IIe can be used with the GS as well. The main exceptions are multifunction cards, such as the Street BusinessCard and the Prometheus Versacard, which simulate the effect of having an I/O device in a particular slot even though the card plugs into a different slot. (These devices are said to use phantom slots.)

The GS also has a unique eighth slot for memory expansion only. Through it you can add up to 8 megabytes of RAM memory as well as additional ROM memory. Although the standard Apple GS Memory Expansion Card actually holds only 1M of RAM, other manufacturers sell cards with greater capacities. (See Appendix 6 for information on available GS memory expansion cards.)

In addition to expansion slots, the GS has the same built-in expansion ports as the IIc, plus a few more. Here are the ones that correspond to standard slot numbers:

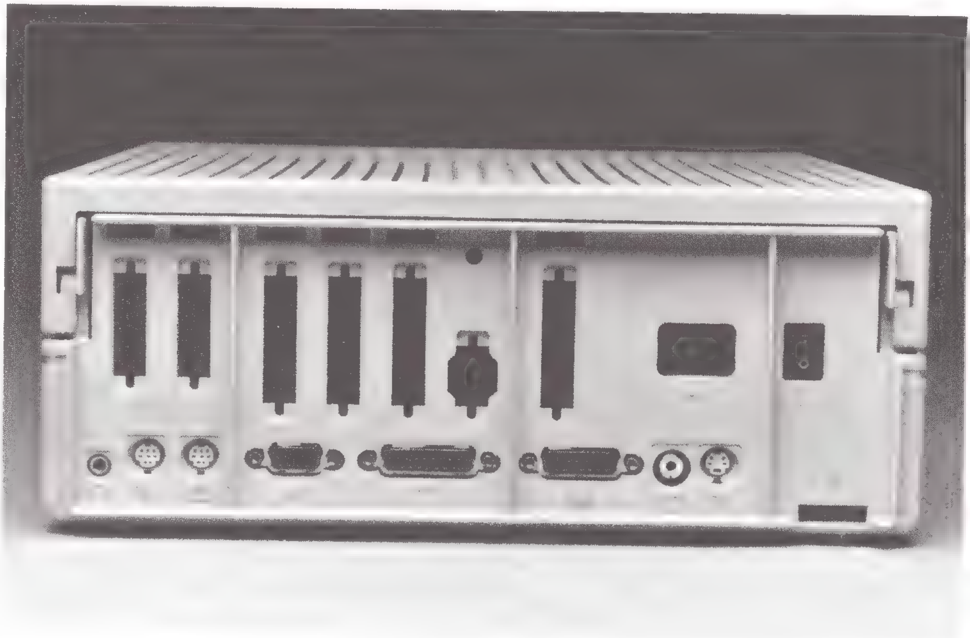
- Port 1: serial printer port

- Port 2: serial modem port

- Port 3: video port

- Port 4: mouse port (part of the Apple DeskTop Bus™)

Photo 1-1: The Back Panel of the Apple IIGS, Showing the I/O Connectors



- Port 5: disk-drive port (SmartPort)
- Port 6: 5¼-inch disk-drive port
- Port 7: AppleTalk (actually uses serial port 1 or 2)

As shown in photo 1–1, the connectors for these ports are located on the back panel of the GS.

The other major I/O ports are two game control ports, a built-in clock/calendar, and a sound synthesizer.

It is important to realize that an internal port and its corresponding slot are mutually exclusive. In other words, you can use only one or the other. It is not possible to switch between an internal port and a plug-in card without turning the GS off and then on again.

To tell the GS whether you want to use a slot or a port, you must use a built-in configuration program called the Control Panel. It is an example of a Classic Desk Accessory (see chapter 9), a utility program you can switch to even when you are using another program.

To bring up the classic desk accessory menu, enter Control-OpenApple-Esc from the keyboard. This pops up a list of desk accessories you can use, beginning with the Control Panel. To activate the Control Panel, use the up- and down-arrow keys to move the inverse bar over its name, and then press Return. You will then see the main command menu for the Control Panel. Most of these commands are for setting global parameters that configure the system the way you want. For example, there are commands to set the date and time, default video characteristics, and keyboard options.

The Slots command is used to set the desired slot/port configuration. Once you select it, you will see a list of slot numbers and an indication of whether a built-in port or "Your Card" (a card in a slot) is to be active. The default set-up selects internal ports 1 through 6 and your card in slot 7. If you wanted to use a parallel printer card in slot 1, you would use the right-arrow key to change the entry for slot 1 to "Your Card." The next time you start up the GS, the card in slot 1 will be used instead of serial port 1.

The parameters set up by the Control Panel are stored in a memory area associated with the GS clock chip (not part of the 65816 address space). This memory area is nonvolatile (because it is powered by the clock's battery), so the GS remembers the Control Panel settings even when the GS is turned off.

RCA Mini Headphone Jack

By plugging headphones into the RCA mini headphone jack, you can divert sound from the GS's on-board speaker to the jack. Although it is a stereophonic jack, the GS sound circuitry sends the same output signal to both channels, so you will not hear true stereophonic sound unless you add an optional sound card.

Serial Ports

The GS has two built-in serial interface ports driven by a Zilog 8530 Serial Communications Controller (SCC), the same chip used on the Macintosh™ computer. By convention, the first port (port 1) should be used with a serial printer, such as the ImageWriter® II, and the second (port 2) should be used with a modem, such as the Hayes Smartmodem. You can also use either port to connect the GS to an AppleTalk network; in this case the serial port you choose behaves like a port 7 device.

The 8530 is not the same as the serial chip used on the IIc and the Apple Super Serial Card. This means that programs for the IIc or Super Serial Card that access serial hardware registers directly will not work on the GS without a new device driver. Most programs that are chip-specific are communications programs. Printing programs do not usually cause problems, because output goes through the firmware that controls the serial port; at this level, the serial ports are largely compatible.

Game I/O Port

The game I/O port is a nine-pin DB-9 connector used to interface game controllers such as joysticks, paddles, and push buttons. It is equivalent to a similar port on the back of the IIe and IIc. With it you can measure the settings of four variable-resistance devices (such as two X–Y joysticks) and three single-bit switch inputs.

The motherboard of the GS also has a rectangular 16-pin game connector that is identical to the one on the IIe. It provides all the electrical signals of the game I/O port as well as four single-bit output annunciator signals for driving indicator lights or simple relays.

Disk-drive Port

The disk-drive port is similar to the one on the back of the IIc. It is an Apple SmartPort interface, which means that you can connect to it any device adhering to the SmartPort hardware and software specifications. This includes 3½-inch drives, such as the UniDisk™ 3.5 and the Apple 3.5 Drive, and 5¼-inch drives, such as the UniDisk 5.25 and the DuoDisk®.

To add disk drives to the GS, all you have to do is daisy-chain them to the SmartPort in the correct order. (See appendix 7 for information on available IIGS disk drives.) The Apple 3.5 Drives must be connected first (two maximum), followed by any number of UniDisk 3.5 drives, and then followed by any 5¼-inch drives (two maximum). Because of power supply limitations, you should not add more than two UniDisk 3.5 drives to the chain.

The GS treats the 5¼-inch drives at the end of a SmartPort device chain as port 6 devices, even though the disk-drive port is internally mapped to port 5. This means that you can boot from the 5¼-inch drive with an Applesoft PR#6 command. If you want to use your 5¼-inch drives with a disk controller card in slot 6 instead, use the Control Panel Slots command to change the slot 6 entry to “Your Card.”

The Control Panel also lets you choose which disk device you want the GS to boot from when you turn it on or when you enter Control-OpenApple-Reset. The default is “Scan,” which tells the GS to look for a drive from slot 7 down to slot 1 and to boot from the first one it finds with a bootable disk. You can also specify a specific slot number to boot from; do this if you want to boot directly from the first 3½-inch disk in port 5 instead of from a 5¼-inch disk in port 6, for example.

Using the Control Panel RAM Disk command, you can also allocate an area of memory for use as a RAM disk device. RAM disk memory contains data that is formatted in the same way as on a real disk. Data is read or written using the same commands as for a standard drive, but I/O operations are much faster because there is no waiting for slow-moving mechanical parts. The main drawback of a RAM disk is that data on the RAM disk disappears when you turn off the power, so you must transfer it to a real disk before turning off the computer.

If the RAM disk has been properly formatted with all the code necessary for booting, you may want to assign it as the boot device instead of a real disk drive. If you do so, the operating system will reload more quickly after a Control-OpenApple-Reset reboot.

The GS also supports any ROM disk device you might have on a memory expansion card. A ROM disk, like a RAM disk, is fast, but its contents are permanent. In a typical case, a ROM disk would contain the GS operating system and any utility or application programs to which you need instant access.

Analog RGB Video Port

You can connect an analog RGB monitor, such as the AppleColor RGB Monitor, to the analog RGB video port to view the 4,096 colors the GS supports in its super high-resolution graphic display mode. You cannot connect digital RGB monitors to this port.

Composite Color Video Port

The GS also generates an NTSC-compatible composite color video output signal. The port connector is an RCA phono connector. You can use the NTSC signal to drive an ordinary monochrome monitor or a composite color monitor.

Apple DeskTop Bus Port

The Apple DeskTop Bus (ADB) is a unique data transmission interface designed to allow you to connect, in a chain, certain types of input devices to the GS. The two most common ADB devices are the keyboard and the mouse, which come with the GS. The ADB protocol is general enough, however, to work with other pointing devices, such as trackballs and joysticks. The GS communicates with any device on the ADB by sending it commands and listening for responses; communication signals can be directed to a particular device to avoid interfering with other devices in the chain.

Connected directly to the ADB is the keyboard. (See photo 1–2). It contains 80 keys, including a 14-key numeric keypad. The layout of keys is almost the same as the layouts on the IIe and IIc keyboards, but with one important difference: the Option key (formerly called the Solid-Apple key) is now on the left side of the space bar, just to the left of the Open-Apple key.

The main new feature of the keyboard is its programmability. By changing Control Panel parameters, you can adjust the auto-repeat rate of the keys and the time delay before repeating begins. You can even make the space bar, Delete key, and arrow keys repeat more quickly than other keys. Another nice feature is the ability to enable keyboard buffering allowing you to type ahead of your program. Finally, you can determine, from a program, whether any key on the keyboard is down, even modifier keys like Shift, Control, and Caps Lock.

The GS keyboard has two ADB connectors, one for attaching it to the ADB port and one for connecting it to the next device in the ADB chain.

Photo 1-2: The Apple IIGS Keyboard

The next device is usually the GS mouse. For compatibility reasons, the GS treats the mouse as the port 4 device, even though it is really part of the ADB chain. The standard GS mouse contains only one button, but the system software supports two-button mice.

REAL-TIME CLOCK/CALENDAR

The GS is the only member of the Apple II family with a built-in real-time clock/calendar chip. With the help of a small battery on the motherboard, this chip keeps track of the current date and time, even after you have turned off the GS.

The newest versions of the ProDOS® operating system automatically recognize the GS clock and will use it to time- and date-stamp files when they are saved to disk. You can also read the clock from a program using two tool set functions in ROM. To set the time, use the Control Panel.

You can add a clock to a IIe, but most of the clocks being sold use up an expansion slot. The GS clock, on the other hand, does not occupy a slot (or a port), so you can save a slot for some other type of peripheral.

VIDEO DISPLAY MODES

The GS supports all the standard video display modes of the IIe and IIc, plus one important new mode. The standard display modes are as follows:

> 80-column text mode. The dimensions of the screen are 80 columns by 24 rows.
>
> 40-column text mode. The dimensions of the screen are 40 columns by 24 rows.
>
> Low-resolution graphics mode. The dimensions of the screen are 40 dots horizontally by 48 dots vertically. Each dot can be one of sixteen colors.
>
> Double-low-resolution graphics mode. The dimensions of the screen are 80 dots horizontally by 48 dots vertically. Each dot can be one of sixteen colors.
>
> High-resolution graphics mode. The dimensions of the screen are 280 dots horizontally by 192 dots vertically. Each dot can be one of six colors; color choices depend on the column in which the dot is found.
>
> Double-high-resolution graphics mode. The dimensions of the screen are 560 dots horizontally by 192 dots vertically. This mode supports sixteen different colors.

The GS adds two minor embellishments to the text modes: the color of text characters and the background color are user-selectable. Use the Control Panel to set them to any of sixteen colors. You can also use the Control Panel to set a screen border color.

The new GS video display mode is the super high-resolution graphics mode. It is the mode used to display the outstanding color graphics images for which the GS has become famous. The super high-resolution screen is 200 dots in height. The horizontal dot resolution is separately controllable for each of the 200 lines, and can be 320 dots or 640 dots. Most programs keep all the lines at the same resolution to simplify graphics operations.

The colors available in super high-resolution graphics mode depend on the horizontal resolution. In the 640-by-200 mode, each dot can be one of four colors selected from a group of sixteen assigned to the line. The foursome available depends on the column position; without using tricky programming techniques, only 256 colors may appear on the screen at once. In the 320-by-200 mode, each dot can be one of sixteen colors; 256 colors may appear on the screen at once.

The colors used in super high-resolution graphics mode are completely programmable and are formed by mixing one shade of red with one shade of green and one shade of blue. Because 16 shades are available for each of these primary colors, a total of 4,096 colors can be generated.

SOUND

The GS has the most advanced sound capability of any microcomputer. Tucked away in a corner of the motherboard is a powerful device, called an Ensoniq™ Digital Oscillator Chip (DOC), which is a professional-quality synthesizer for music, speech, and sound. This same chip controls the popular Mirage Music Synthesizer used by musicians around the world.

By feeding the Ensoniq chip the right software, you can create amazing audio effects, from symphonic sound to your own voice. The chip contains 32 oscillators for stepping through audio waveform tables, so theoretically you can play back 32 notes simultaneously. In the GS implementation, oscillators are paired off to create 15 generators or voices (the 16th generator is reserved for internal use); this is still enough to create superb sound quality.

You can play back monophonic sound on the GS through a two-inch on-board speaker or through the RCA mini headphone jack on the back panel. If you plug in headphones, the speaker is automatically disabled. To get stereo sound, you can add an inexpensive hardware demultiplexer, such as the SuperSonic card from MDIdeas. Such a device samples the audio waveform output and the channel number (from 1 to 8). Stereo sound is then created by diverting the odd-numbered channels to one speaker and the other channels to the other speaker. Of course, it is up to the sound-generation software to direct the outgoing signal to different channels.

The GS has a 64K RAM area that is used for storage of waveforms. This RAM is not part of the 65816 address space, but can be read from or written to through a hardware interface called the Sound General Logic Unit (GLU).

The Ensoniq is also able to sample incoming analog waveforms and store them, in digital form, in its 64K RAM area. This is done by attaching the incoming sound source, such as a microphone, to an analog input connector and then triggering analog-to-digital conversions by sending commands to the Ensoniq chip.

OPERATING SYSTEM SOFTWARE

Over the past ten years or so, Apple has developed and released four major disk operating systems for the Apple II family, all of which work on the GS:

- DOS (1978)
- UCSD Pascal (1979)
- ProDOS 8 (1983)
- ProDOS 16 (1986)

Other operating systems are available from third-party developers, but only CP/M seems to generate much interest. CP/M does not work with a plain GS; you will need to install a Z-80 coprocessor card first.

DOS 3.3, the latest (and final) version of Apple's first operating system, is no longer actively promoted by Apple. Nevertheless, it continues to be popular because there is a large body of software which works with it. DOS 3.3 does not take advantage of any special features of the GS, and it does not work (without modification) with 3½-inch disk drives or hard drives; it was designed to work only with 5¼-inch drives and their associated 140K floppy disks.

The UCSD Pascal operating system is primarily for those who wish to program in Pascal. At one time, Pascal seemed to be the language of choice for educators because students are able to learn the fundamentals of a structured programming language quickly by writing a few programs in Pascal. Unfortunately, UCSD Pascal has never received the widespread support which DOS 3.3 or ProDOS has. It is certainly rare to find a professionally published program for the Apple II written in Pascal.

Apple released ProDOS 8 (it was originally called just "ProDOS") to solve the problem of using high-capacity disks with the Apple II family. Originally, this meant hard disks like the Apple ProFile™, but the category now includes the 800K 3½-inch disks used by the Apple 3.5 Drive and the UniDisk 3.5. ProDOS 8 works nicely with disk volumes up to 32 megabytes and with files up to 16 megabytes. In addition, it uses a hierarchical directory structure to make it easier to manage the hundreds of files which can be stored on high-capacity disks.

ProDOS 16 is the only Apple II operating system that works only on the GS. Although it uses the same disk file storage format as ProDOS 8, you must use different programming techniques to control it. This means ProDOS 8 applications will not work under ProDOS 16, and vice-versa; however, you can store files created by either operating system on the same disk.

Apple developed ProDOS 16 for the GS because ProDOS 8 works only with programs that run in the first 64K of memory. This is an unreasonable restriction in a system, like the GS, that has a 16-megabyte address space. With ProDOS 16, you can issue disk commands from anywhere in memory.

ProDOS 16 is described in greater detail in chapter 10. For an in-depth analysis of ProDOS 8, refer to the book *Apple ProDOS: Advanced Features for Programmers* (Little, 1985). Other useful books are listed in appendix 8.

DEVELOPMENT SOFTWARE

The official development software for the GS is the Apple Programmer's Workshop (APW). It runs under ProDOS 16 and is distributed on 800K disks, so you will need at least one 3½-inch disk drive to use it.

APW is made up of several integrated modules accessed through a common command shell:

- A text editor for creating program source code
- A linker for combining object code modules created by APW-compatible assemblers and compilers
- A 65816 assembler for compiling assembly language programs
- A 65816 debugger for tracking down errors in programs

An APW-compatible C compiler is also available from Apple. Similar compilers for Pascal and BASIC are available from third parties or are in development. (See appendix 8.)

The APW shell supports many commands that will help you perform common software development tasks, such as moving files from disk to disk, renaming files, deleting files, and displaying the names of on-line volumes.

CREDITS

So who was responsible for designing the GS, anyway? The answer is as close as your keyboard. Remove the disk from your start-up drive and turn on (or reboot) the GS. A "Check startup device!" error message will appear on the screen along with an apple icon that slides back and forth. Now for the secret command: while holding down the Open-Apple and Option keys, type Control-N. This tells the GS to reveal the names of all the key people who worked on the GS project. You read it here first.

CHAPTER 2

Programming the 65816 Microprocessor

The 65816 microprocessor that controls the GS can operate in either emulation mode or native mode. In emulation mode, the 65816 understands the same instructions (and some new ones, too) as the 65C02 microprocessor in the Apple IIc and the enhanced Apple IIe. Because the GS hardware can emulate the characteristics of the IIe and IIc almost exactly, this means most IIe and IIc software will work on the GS without modification.

Note: For convenience, consider future references to the IIe as references to the IIc as well.

In native mode, the 65816 supports many new and important features that make it substantially more powerful than a 65C02:

- A directly addressable 16-megabyte address space
- 16-bit accumulator and index registers
- 16-bit memory operations
- A relocatable stack and direct page
- Many extra instructions and addressing modes

By exploiting these features, developers can create powerful programs much more easily than they could on a IIe with a 65C02. Refer to the manufacturer's 65816 data sheet in appendix 2 for a detailed description of the microprocessor.

The 65816 in the GS is able to operate at either of two speeds: the normal 1.0 MHz rate of the IIe or the faster rate of 2.8 MHz. (Overhead for RAM refreshing actually reduces the average fast rate from 2.8 MHz to 2.5 MHz for RAM operations; ROM operations take place at full speed.) The faster speed is the default, and most programs will run properly at that speed. Some IIe applications, however, notably those that rely on precise timing loops (games and simulators, for example), may work only at normal speed. You can change the current operating speed with the Control Panel System Speed command.

This chapter investigates how the 65816 operates. This includes an examination of its internal registers, its instruction set and addressing modes, and two important data areas called the stack and direct page. Methods of creating 65816 programs using the APW assembler are also covered. Throughout the chapter, the emphasis is on features unique to the 65816 running in native mode, but references to 65C02 emulation mode features are made where necessary.

THE 65816 MEMORY SPACE

The 65816 has a 24-bit address bus, meaning it can access an enormous 16-megabyte (2^{24} bytes) memory space. Internally, it deals with this memory as a series of 64K banks, numbered from $00 to $FF. A standard GS has RAM in banks $00, $01, $E0, and $E1 and ROM in banks $FE and $FF. Chapter 4 discusses exactly how the GS makes use of its memory space.

The first bank, bank $00, has special qualities. It contains two interesting data areas called the *direct page* and the *stack*. In native mode, you can place these areas anywhere in bank $00. In emulation mode, the stack must be in page $01; the direct page can be any page in bank $00, but is invariably page $00 (zero page). (A page is a group of 256 consecutive bytes beginning at an address whose last two digits are $00.)

Direct Page

The 256-byte direct page is important because those 65816 instructions that make use of it are shorter and faster than similar instructions using other parts of memory. That is because the program needs to provide only one byte of address information to the 65816 (the offset into the direct page); the other two bytes needed to form a complete 24-bit address are taken from an internal register.

More importantly, the 65816 has powerful indirect indexed addressing modes that require you to place a pointer to a data structure in direct page. To access a field in the data structure, all you have to do is specify the direct page location and the offset to the field. Unlike some microprocessors (notably the 68000), the 65816 does not allow you to access a data structure indirectly by putting its address in a microprocessor register; this makes it necessary to have a data area such as a direct page.

When you are developing programs in emulation mode, it is common to refer to direct page as *zero page*, because it should be located at page $00 in bank $00, as it is when using the 65C02.

Stack

The stack is an area of memory that is implicitly used by certain 65816 instructions to store data. This area is "inverted" in the sense that it grows down in memory as you add data to it (a "push" operation) and shrinks up in memory as you remove data from it (a "pull" operation).

The active position in the stack is given by the value in a 16-bit stack pointer (SP) register, described below. It is initially set to the last byte (highest-addressed byte) in the stack area.

It is important to understand the mechanics of push and pull operations, because they are used quite often. When you push a byte value on the stack, the byte is placed at the address that is in the stack pointer and the stack pointer is decremented by one. When a byte is pulled from the stack, the stack pointer is incremented and the value is loaded from the new address it contains. The important point to realize is that the stack pointer always points to the byte just below the last item pushed on the stack.

In native mode, the stack could occupy the whole of bank $00, although most programs never need more than a page or two. In emulation mode, the stack is always 256 bytes long and occupies page $01 of bank $00.

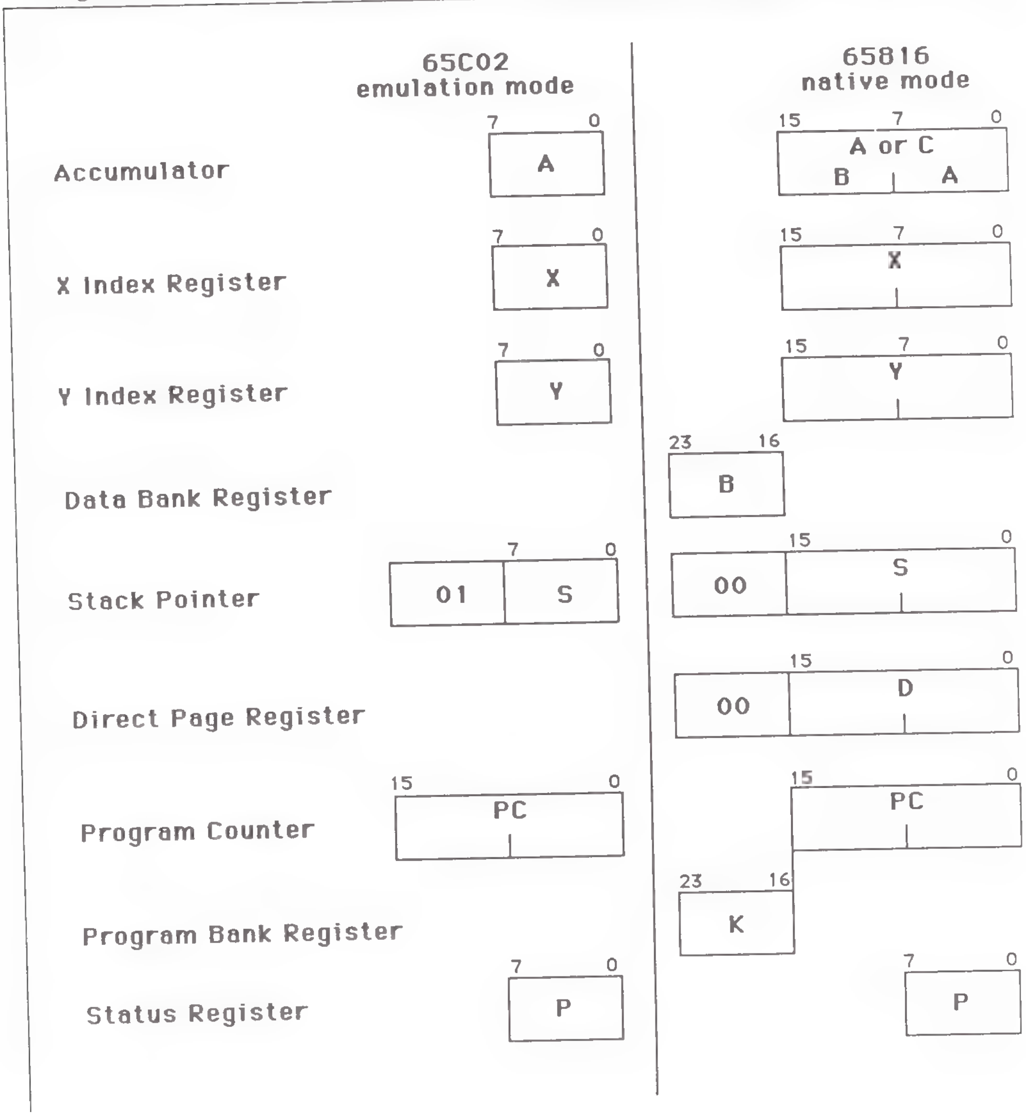
THE 65816 REGISTERS

Registers are areas inside a microprocessor that the microprocessor uses to store values and manipulate data. They are not memory locations in the microprocessor's address space. In general, the more registers a microprocessor has, the more powerful the microprocessor is; that is because operations involving registers are much faster than similar operations involving memory locations.

The registers used by the 65816 are shown in figure 2–1. There are nine of them:

- Accumulator
- X index register
- Y index register
- Data bank register
- Program bank register
- Direct page register
- Processor status register
- Stack pointer
- Program counter

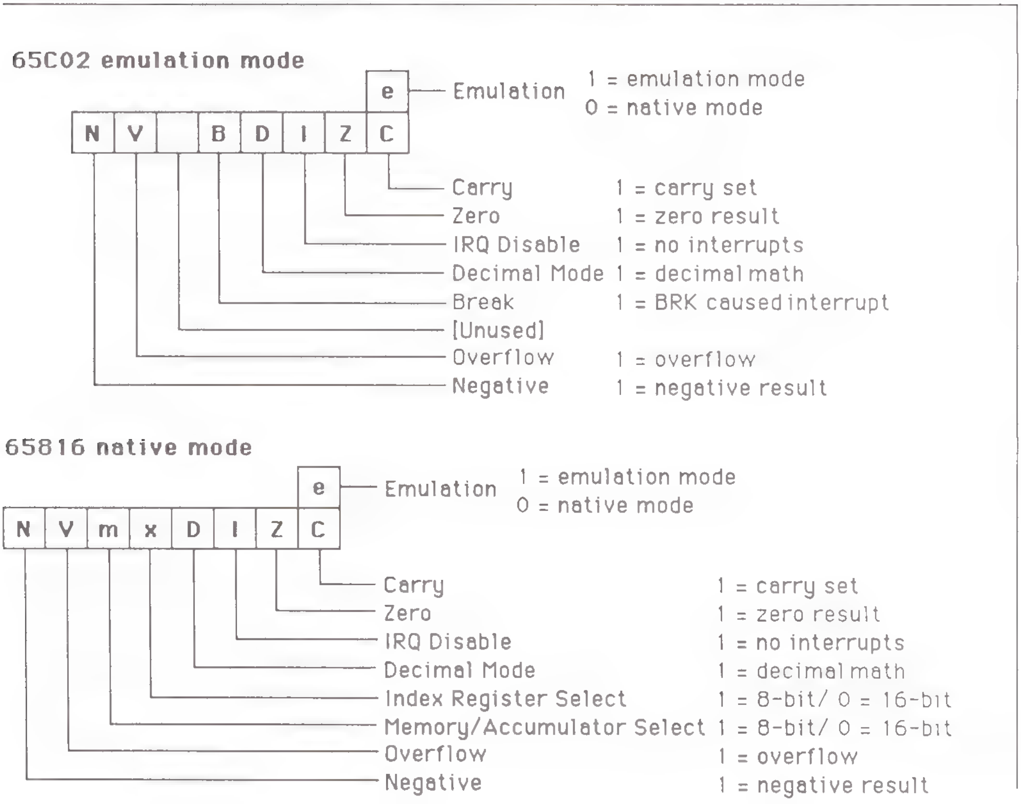
Figure 2–1. The 65816 Register Sets in Emulation Mode and Native Mode



Processor Status Register

The processor status register (P) contains a group of status flags and mode select bits that reflect the operational state of the 65816 (see figure 2–2). Most instructions cause one or more status flags to change according to the operations they perform;

Figure 2–2. The Status Flags in the 65816 Processor Status Register


65C02 emulation mode
e
Emulation
1 = emulation mode
0 = native mode
N V B D I Z C
Carry 1 = carry set
Zero 1 = zero result
IRQ Disable 1 = no interrupts
Decimal Mode 1 = decimal math
Break 1 = BRK caused interrupt
[Unused]
Overflow 1 = overflow
Negative 1 = negative result
65816 native mode
e
Emulation
1 = emulation mode
0 = native mode
N V m x D I Z C
Carry 1 = carry set
Zero 1 = zero result
IRQ Disable 1 = no interrupts
Decimal Mode 1 = decimal math
Index Register Select 1 = 8-bit/ 0 = 16-bit
Memory/Accumulator Select 1 = 8-bit/ 0 = 16-bit
Overflow 1 = overflow
Negative 1 = negative result

status-checking instructions can then be used to change the flow of a program based on the result.

The structure of the status register depends on whether the processor is in emulation mode or native mode, as figure 2–2 indicates. The flags in the emulation mode status register are the carry flag, the zero flag, the IRQ disable flag, the decimal mode flag, the break flag, the overflow flag, the negative flag, and the emulation flag. In the native mode status register, the break flag is not present, the spare status bit is used, and two register-select flags are available: the memory/accumulator select flag and the index register select flag. The flags in the emulation mode status register and the native mode status register are discussed below.

Carry Flag. The carry flag is most often used when performing addition (ADC) or subtraction (SBC) operations. As is shown later in this chapter, this flag must be cleared (with CLC) before adding two numbers and must be set (with SEC) before subtracting two numbers.

The carry flag is also used to check the result of a comparison operation (CMP, CPX, CPY). If the number in the specified register is less than the number it is being compared to, the carry flag is cleared; otherwise it is set. The BCC and BCS branch-on-condition instructions can then be used to switch control to another part of the program depending on the result.

The carry flag also participates in the shift and rotate instructions: ASL, LSR, ROL, and ROR. These instructions are described later in this chapter.

Zero Flag. The zero flag indicates whether the result of the last arithmetic operation or register loading operation was zero. If it was, the zero flag is set to 1; otherwise, it is cleared to 0.

IRQ Disable Flag. The IRQ disable flag controls how the 65816 reacts to hardware interrupt request (IRQ) signals. (Interrupts are discussed later in this chapter.) If the flag is set to 1, IRQ interrupts are ignored; otherwise the 65816 processes them.

Decimal Mode Flag. This flag controls the types of numbers the 65816 uses to perform addition (ADC) and subtraction (SBC) operations. When the decimal mode flag is off, standard binary arithmetic rules are used. This is the mode in which most programs operate and it is the default mode when the GS is turned on.

When decimal mode is on, the 65816 assumes all numbers are stored in binary-coded decimal format and generates arithmetic results that are in this format as well. In this format, each byte contains exactly two digits from 0 to 9; the most-significant digit occupies the high-order four bits of the byte.

Break Flag. The break flag indicates whether the source of a 65816 interrupt in emulation mode is a BRK instruction (the flag is set) or a hardware interrupt request (the flag is clear). Unlike native mode, BRK and interrupt requests vector to the same subroutine; the subroutine can inspect the break flag to determine the cause of the interrupt.

Overflow Flag. The overflow flag usually indicates whether the result of an arithmetic operation involving two's-complement (signed) numbers is within prescribed limits. The 65816 uses the two's-complement form for signed arithmetic operations to simplify internal calculations. The limits are -128 to +127 for 8-bit numbers and -32768 to +32767 for 16-bit numbers.

The most-significant bit of a two's-complement number is the sign bit: it is 0 for positive numbers and 1 for negative numbers. For positive numbers, the remaining bits represent the magnitude of the number in standard binary form.

Things are not quite so simple for negative numbers. To determine the two's-complement form, take the absolute value of the number, complement it, then add 1 to the result. Here is how to form -22, for example:

```
00010110    (+22 in binary)
11101001    (+22 complemented)
       1    (add 1)
--------
11101010    (-22 in two's-complement form)
```

Notice that the high-order bit is set, identifying this as a negative number.

Negative Flag. This flag reflects the value of the high-order bit of the last value transferred directly to the A, X, or Y register or put there as a result of a calculation. The high-order bit is bit 7 for an 8-bit register or bit 15 for a 16-bit register.

The negative flag derives its name from the fact that the high-order bit of a two's-complement number reflects its sign: if the bit is 1, the number is negative.

Emulation Flag. The emulation flag hides behind the carry flag in the status register. You will use it to set the operating mode of the 65816. When it is set to 1, the 65816 operates in 65C02 emulation mode; when it is 0, the 65816 operates in native mode.

Entering native mode paves the way to powerful 16-bit register and memory operations, as is discussed below.

The only way to affect the emulation bit is to set or clear the carry flag (with SEC or CLC) and then execute the XCE (exchange carry with emulation) instruction.

Register Select Flags. There are two register select flags in the native mode status register: the memory/accumulator select flag and the index register select flag.

The index register select bit is called *x* and the memory/accumulator select bit is called *m*. When *x*=1 the 65816's X and Y registers are both 8 bits in size; when *x*=0, these index registers are 16 bits. Similarly, when *m*=1 the A register is 8 bits in size and all memory operations (such as DEC, INC, ASL, and others) involve one byte only. When *m*=0, the A register is 16 bits in size and memory operations act on two consecutive bytes (a word).

When the *m* and *x* bits are both 0, the 65816 is said to be in full native mode. This is the mode you will usually want to use when you develop a GS-specific application.

The 65816 has two special instructions you can use to set or clear the *m* and *x* bits: REP and SEP. To clear bits, use the REP instruction:

```
   REP      #%00110000       ;clear m and x bits
```

Note: As is discussed later in this chapter, the # symbol means the number is a constant, not an address; % identifies the number as a binary number.

REP clears to 0 those bits in the status register that correspond to 1 bits in the operand. (The operand is the number following REP.) Thus, you would use REP to activate 16-bit register/memory and indexing operations.

Conversely, SEP sets to 1 those bits in the status register that correspond to 1 bits in its operand. Use SEP to revert to 8-bit register/memory and indexing operations.

Accumulator

The accumulator (A) is an 8-bit or a 16-bit register, depending on the state of the *m* flag in the native mode status register. (Actually, even in 8-bit mode you can access the "other" 8 bits of the accumulator with the XBA exchange instruction.) In emulation mode, the accumulator is always 8 bits in size.

The accumulator is the only register that works with the 65816's addition (ADC) and subtraction (SBC) instructions. In fact, that is how it gets its name: it accumulates the results of arithmetic operations. It is also the only register which participates in logical operations (ORA, AND, XOR).

Most instructions refer to the accumulator by the symbol A, no matter what its size. Some call it C to emphasize that they act on the entire 16 bits even if the *m* flag is 1. The two halves of the 16-bit registers are also referred to as A (low 8 bits) and B (high 8 bits). This is for the benefit of the XBA instruction, which swaps the two halves of the 16-bit accumulator.

Using a 16-bit accumulator is convenient because it can handle numbers up to 65,535 (instead of just 255), thus simplifying many types of arithmetic instructions.

A 16-bit accumulator is somewhat less convenient for dealing with character strings, however, because you usually want to load only one character into the accumulator at once. In these situations, you may want to temporarily revert to an 8-bit register using the SEP #%00100000 instruction.

Index Registers

The index registers are called X and Y. They are commonly used to hold offsets into a data structure from a fixed reference location; hence the name *index*.

Like the accumulator, the index registers are 8 bits in size in emulation mode. In native mode, they can be either 8 bits or 16 bits in size: if the *x* flag is 0, they are 16-bit registers; if it is 1, they are 8-bit registers.

Note that the *x* flag always affects both index registers: it is not possible to have X and Y registers that are different sizes.

Stack Pointer

The 16-bit stack pointer register (SP) points to an address in bank $00 that is one byte below the address of the last value pushed on the stack. It automatically decreases after push instructions, such as PHA and PHX, and increases after pop instructions, such as PLA and PLX.

In emulation mode, the high-order byte of the stack pointer is always $01, so the stack is confined to 256 bytes.

Direct Page Register

The 16-bit direct page register (D) contains the address of the start of the direct page in bank $00. It can contain any value, but direct page operations are faster if direct page begins on a page boundary. In emulation mode, the direct page register should be set to $00.

Data Bank Register

The 8-bit data bank register (B) indicates the default memory bank for memory access operations. When you specify a 16-bit address in an instruction, the 65816 calculates the full 24-bit address by concatenating the data bank register. In emulation mode, the data bank register should be set to $00.

Program Bank Register

The program bank register (K) is an 8-bit register that indicates in which memory bank the current program is operating. It combines with the 16-bit program counter register to mark the exact position in memory of the instruction currently being executed.

There is no explicit instruction for changing the program bank register. It implicitly changes when the program transfers control to another bank with a JSL or JMP instruction.

In emulation mode, the program bank register should be set to $00.

Program Counter

The 16-bit program counter (PC) holds the offset from the start of the current program bank of the instruction currently being executed. The 65816 automatically increments the PC as it processes instructions, usually by adding the size of the instruction just executed. When a flow of control instruction such as JMP or BRA is encountered (see below), however, the operand address is put into the PC, causing execution to continue at the target address.

Note that when the PC changes from $FFFF to $0000 at a memory bank boundary, the program bank register is not incremented. This means that control passes to the beginning of the current bank, not to the beginning of the next bank. As a result, a 65816 program is not permitted to cross a bank boundary, although it can call a program or subroutine in another bank. On the GS, the ProDOS 16 program loader always loads a program into a single bank.

THE 65816 INSTRUCTIONS

The 65816 supports 91 different instructions. (See appendix 2 for detailed descriptions of each.) This includes every 65C02 instruction as well as 27 instructions unique to the 65816. Each instruction may use one or more addressing modes to locate data it uses; the number of instruction/addressing mode combinations is 256. You will find a description of addressing modes in the next section.

A one-byte binary opcode tells the 65816 what the instruction is and what addressing mode it is using. You do not have to memorize these opcodes, because you will use an assembler to convert mnemonic instructions and addressing modes in a source code file to binary object-code form.

Following the opcode there may be up to three operand bytes describing a number or an address to be used by the instruction. Multibyte operands are always stored with the low-order byte first. Again, the assembler takes care of arranging these bytes properly for you.

The next section of this chapter examines the 65816 instructions, discussing what they do, and how they should be used. Refer to table R2-1 (in the reference section at the end of the chapter) for an alphabetical list of instructions. This table also contains the permitted addressing modes and opcodes for each instruction type as well as the number of cycles the 65816 uses to complete each instruction.

(The time for one cycle is the reciprocal of the clock speed. The clock speed is either 2.8 MHz or 1.0 MHz, depending on the setting of the Control Panel System Speed option. For programs operating in RAM in fast mode, the average clock speed is actually about 2.5 MHz because of overhead introduced by RAM refreshing requirements.)

Note: Instructions marked with an asterisk in the following sections are not supported by the 65C02 microprocessor.

Load/Store Instructions

LDA	Load the accumulator (byte or word)
LDX	Load the X register (byte or word)
LDY	Load the Y register (byte or word)
STA	Store the accumulator (byte or word)
STX	Store the X register (byte or word)
STY	Store the Y register (byte or word)
STZ	Store zero to memory (byte or word)

The first three instructions, LDA, LDX, and LDY, let you load 8- or 16-bit values into the accumulator and index registers. The data size depends on the settings of the *m* and *x* bits in the 65816 status register. The value can be a specific number or a number stored at a given memory location.

The STA, STX, and STY instructions let you transfer the contents of a register to a particular memory location. STZ stores a zero at a memory location.

When loading word-sized values from a given address, keep in mind that the low-order 8 bits of the register are filled with the number stored at Address and the high-order 8 bits are filled with the number stored at Address+1 (the next higher address). Similarly, when storing words, the low-order 8 bits of the register are stored first.

Push Instructions

*PEA	Push effective absolute address (word)
*PEI	Push effective indirect address (word)
*PER	Push effective relative address (word)
PHA	Push the accumulator (byte or word)
*PHB	Push the data bank register (byte)
*PHD	Push the direct page register (word)
*PHK	Push the program bank register (byte)
PHP	Push the status register (byte)
PHX	Push the X register (byte or word)
PHY	Push the Y register (byte or word)

Use the push instructions to transfer data to the top of the stack and decrement the stack pointer by the data size. The data size is either byte or word, depending on the instruction and the settings of the *m* and *x* status bits. Word operands are pushed high-order byte first.

PHA, PHB, PHD, PHK, PHP, PHX, and PHY all push the contents of a register on the stack. PEA, PEI, and PER push two-byte addresses, as follows:

> PEA pushes an absolute 16-bit address; most programmers really use it to push numeric constants, however, because it does not involve loading the constant into a register first.
>
> PEI pushes the address stored in two consecutive direct page locations.
>
> PER pushes the address, which is a specified number of bytes from the position of the instruction. This instruction is useful if you are developing relocatable code (code that runs at any load address).

The operand for a PEA, PEI, or PER instruction gives the 65816 the address it needs to work with.

Pull Instructions

PLA	Pull the accumulator (byte or word)
*PLB	Pull data bank register (byte)
*PLD	Pull direct page register (word)
PLP	Pull status register (byte)
PLX	Pull X register (byte or word)
PLY	Pull Y register (byte or word)

These instructions transfer data from the top of the stack to a register, then increment the stack pointer by one (byte operation) or two (word operation). Word operands are popped low-order byte first.

Inter-register Transfer Instructions

TAX	Transfer A to X
TAY	Transfer A to Y
TSX	Transfer S to X
TXS	Transfer X to S
TXA	Transfer X to A
TYA	Transfer Y to A
*TCD	Transfer C to D
*TDC	Transfer D to C
*TCS	Transfer C to S
*TSC	Transfer S to C
*TXY	Transfer X to Y
*TYX	Transfer Y to X

Note: D = data bank register; C = 16-bit accumulator

The inter-register transfer instructions let you move data from register to register. Special reasons for wanting to do this are to initialize the stack pointer (TXS, TCS) and the direct page register (TCD).

Be careful when using the first six instructions in the list if you are transferring between registers of different sizes. For TXA and TYA, if $x=1$ and $m=0$, the high-order 8 bits of the accumulator (B) is zeroed; if $x=0$ and $m=1$, B is not affected. For TAX and TAY, if $m=1$ and $x=0$, the B register is transferred to the upper 8 bits of the index register; if $m=0$ and $x=1$, the upper 8-bits of the index register are not affected. For TXS, the high-order stack register byte is zeroed if $x=1$.

Exchange Instructions

*XBA	Exchange B and A
*XCE	Exchange carry and emulation bits

The two exchange instructions involve manipulating bits inside registers. XBA swaps the upper half of the 16-bit accumulator (B) with the lower half. XCE swaps the carry bit of the status register with the emulation bit so that you can switch between native and emulation mode, as explained earlier in this chapter.

Block Move Instructions

*MVN	Move block in negative direction
*MVP	Move block in positive direction

The block move instructions are powerful instructions that let you transfer a block of data from any part of memory to any other (at least if you are in full native mode). For MVN, X and Y must contain the starting addresses of the source and destination blocks; the length of the block, minus one, must be in the 16-bit C register (even if the *m* flag is not 0). The source and destination banks are specified in the operand for the instruction. MVN moves data from the source block to the destination block in increasing address order.

If the source and destination blocks overlap and the destination block begins at a higher address, do not use MVN, because the end of the source block will be overwritten before its data is moved. In this situation, use the MVP instruction instead. It moves data from the end of the block to the beginning. (Of course, you would not use MVP if the blocks overlap and the destination block begins at a lower address.)

To use MVN, place the ending address of the source and destination blocks in X and Y; as with MVP, the count minus one goes in C.

Flow-of-control Instructions

BCC	Branch if carry flag is clear
BCS	Branch if carry flag is set
BEQ	Branch if zero flag is set (equal)
BMI	Branch if negative flag is set (minus)
BNE	Branch if zero flag is clear (not equal)
BPL	Branch if negative flag is clear (plus)
BRA	Branch always
*BRL	Branch always (long form)

BVC	Branch if overflow flag is clear
BVS	Branch if overflow flag is set
*JML	Jump to long address
JMP	Jump to absolute address
*JSL	Jump to subroutine long, saving return address
JSR	Jump to subroutine, saving return address
*RTL	Return from subroutine long
RTS	Return from subroutine

When the 65816 executes a program, it usually processes instructions in the order in which they appear in the program. You can use the flow of control instructions to tell the 65816 to jump or branch to any position in a program, either unconditionally or only if a certain condition is true.

JML, JMP, BRA, and BRL are unconditional instructions; they always direct the 65816 to another portion of the program. JMP transfers control to a specific memory address. BRA and BRL perform branching relative to the current address in the program counter. The range of a BRA branch is -128 to +127 bytes from the start of the next instruction in the program. BRL branches are from -32768 to +32767, so you can use BRL to move to any part of the current program bank.

The 65816 also supports conditional branching, in which the decision on whether to change the flow of control depends on the setting of flags in the status register. These are the BCC, BCS, BEQ, BMI, BNE, BPL, BVC, and BVS instructions. Use them after any operation that affects the flags so that you can take different actions depending on the result. Refer to table R2-1 to determine how each instruction affects the processor status flags.

For example, after a CMP (compare) command, the carry flag is set if the number in the accumulator is greater than or equal to the number in the operand. Thus, you could use a BCS instruction to transfer control to the portion of the program that deals with this situation.

If a program needs to perform the same task again and again, it is best to convert the task to a callable subroutine so that its code does not have to be repeated. To call a subroutine you can use the JSR or JSL instructions, depending on whether the subroutine ends with RTS or RTL. If the subroutine is in another code bank, you must use JSL.

Both JSR and JSL push on the stack the address, minus one, of the next instruction in the program. For JSR this is an offset from the start of the current program bank (two bytes); for JSL it is a long absolute address (three bytes). The subroutine must end with either an RTS instruction (if called with JSR) or an RTL instruction (if called with JSL). RTS and RTL pop the return address put on the stack by JSR and JSL, put it in the program counter (JSL also updates the program bank register), and increment the program counter. This causes execution to resume at the instruction following the JSR or JSL.

Arithmetic Instructions

ADC	Add memory to accumulator with carry
CMP	Computer accumulator with memory
CPX	Compare X register with memory
CPY	Compare Y register with memory
DEC	Decrement the accumulator or memory
DEX	Decrement the X register
DEY	Decrement the Y register
INC	Increment the accumulator or memory
INX	Increment the X register
INY	Increment the Y register
SBC	Subtract memory from accumulator with carry

You can use INC, INX, and INY to add one to the value in the accumulator or an index register. (INC also works with memory locations.) The largest number a register can hold is $FF (8 bit) or $FFFF (16 bit); the register rolls to $00 if you increment it in these situations.

The corresponding decrement instructions are DEC, DEX, and DEY. They roll the contents of the registers backwards through 0 to $FF (8 bit) or $FFFF (16 bit).

You can add or subtract numbers other than one from the accumulator with the ADC and SBC instructions. To add with CLC, you must remember to clear the carry flag first and to add multiword (or multibyte) numbers starting with the low-order word. For example, to add $1AD88 to the long word stored from Address to Address+3, use code like this:

```
CLC
LDA     Address          ;(assume 16-bit A)
ADC     #$1AD88          ;low word
STA     Address
LDA     Address+2
ADC     #^$1AD88         ;^ = high word
STA     Address+2
```

Subtraction is similar to addition except that you must start with a SEC instruction before performing a series of SBC operations.

The compare instructions, CMP, CPX, and CPY, are really subtraction instructions in disguise. The difference is that the result of the subtraction is not stored anywhere; they simply cause flags in the status register to change depending on the result. The flag settings are as follows:

> The zero flag is set if the two numbers are the same.
>
> The carry flag is set if the (unsigned) number in the register is greater than or the same as the number being compared.

The carry flag is clear if the (unsigned) number in the register is less than the number being compared.

You can use the branch-on-condition instructions to change the flow of control after a comparison operation.

Logical Instructions

AND	Logical AND
EOR	Logical exclusive-OR
ORA	Logical OR

Logical instructions combine the value in the accumulator with the value in the operand in accordance with the rules of Boolean algebra. The result is stored in the accumulator.

For AND operations, all bits in the accumulator that correspond to 0 bits in the operand are cleared to 0; other bits are unaffected. With ORA, all bits in the operand that correspond to 1 bits in the operand are set to 1; other bits are unaffected. The EOR operation is more complicated than either AND or ORA. It sets to 1 all bits that are 1 in the accumulator and 0 in the operand or vice-versa. Bits that are the same in value are cleared to 0.

Bit-manipulation Instructions

BIT	Test bits
TRB	Test and reset bits
TSB	Test and set bits

The BIT instruction performs two functions at once. First, it transfers the high-order two bits of the operand to the negative and overflow flags in the status register (but only if the operand is not an immediate number). If these two bits store program flags, you can follow the BIT with BMI to branch if the high-order bit is set, or with BVS if the next-to-high-order bit is set.

The second function of BIT is to logically AND the contents of the accumulator with the value of the operand. The result is not stored, but the zero flag is conditioned by the operation. This is a handy way of determining if certain bits are clear or set without destroying the contents of the accumulator. For example, to test bits 0 and 1, use the instruction:

```
    BIT #%00000011
```

If either of the last two bits in the accumulator are 1, the zero flag is not set and a subsequent BNE branch will succeed.

You can set or clear any group of bits in an operand with the TRB and TSB instructions. TRB clears to 0 those bits in the operand that correspond to 1 bits in the accumulator. Conversely, TSB sets these bits to 1. TRB and TSB also logically AND the accumulator with the operand and set the zero flag according to the result. The result itself is not stored.

Shift-and-rotate Instructions

ASL	Arithmetic shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right

The shift and rotate instructions let you manipulate the bits in the accumulator or in a memory location by shifting them left or right.

ASL moves the bits in the operand one position to the left. The high-order bit moves into the carry flag of the status register and a 0 appears in bit 0 of the operand. ROL is the same as ASL, except that it is the initial value of the carry flag that is put in bit 0.

LSR moves bits in the operand one position to the right, with bit 0 moving into the carry flag. A 0 appears in the high-order bit of the operand. ROR is the same, except that the initial value of the carry flag is put in the high-order bit.

The operand size for shift and rotate instructions is either 8 bits or 16 bits, depending on the status of the *m* bit in the status register.

System-control Instructions

BRK	Break (software interrupt)
RTI	Return from interrupt
NOP	No operation
SEC	Set carry flag
CLC	Clear carry flag
SED	Set decimal flag
CLD	Clear decimal flag
SEI	Set interrupt flag
CLI	Clear interrupt flag
CLV	Clear overflow flag
*SEP	Set status register bits
*REP	Reset status register bits
*COP	Co-processor (software interrupt)
*STP	Stop the clock
*WAI	Wait for interrupt
*WDM	[Reserved for expansion]

The system control instructions perform a variety of miscellaneous operations. One group (SEC, CLC, SED, CLD, SEI, CLI, CLV, SEP, and REP) can be used to explicitly set or clear bits in the status register. The main use of SEP and REP is to set and clear the *m* and *x* bits in the status register, as explained above.

NOP is a "do-nothing" instruction that simply acts as a place-holder in a debugging operation or a time-waster in a timing loop.

STP, WAI, and WDM are instructions you will probably never need to use. STP essentially shuts down the 65816 until a system reset occurs. WAI puts the 65816 into a low-power mode and then waits until a 65816 interrupt occurs before passing control to the next instruction in the program. The WDM instruction is reserved for future expansion of the 65816 instruction set.

The BRK, COP, and RTI instructions are related to 65816 interrupts and will be discussed later in this chapter.

THE 65816 ADDRESSING MODES

An addressing mode describes the technique the 65816 uses to locate the data with which an instruction needs to work. In a 65816 program source file, the addressing mode information appears in the operand field, just after the instruction mnemonic. The assembler format for each addressing mode is shown in table 2–1.

Knowing the addressing mode, the 65816 can use the operand to calculate an effective address (EA) for the instruction. For read operations, the EA is the address from which data is read; for write operations, it is where data is stored. The effective address calculation varies with the addressing mode, of course: it may be a simple absolute address, an address calculated by adding the contents of the X register to a base address, an address pointed to by two locations in direct page, and so on. The availability of so many different addressing modes makes it easier to develop efficient programs.

This section investigates the various addressing modes you can use with the 65816. Keep in mind, however, that not all addressing modes can be used with every instruction. To see if a particular combination is permitted, refer to table R2–1.

Implied

Many instructions do not require an explicit indication of addressing mode because the instruction mnemonic completely describes what needs to be done. These instructions use the implied addressing mode.

Examples of such instructions are as follows:

```
SED     NOP     TAX     TSX     WAI
CLV     CLD     TAY     TXA     WDM
SEI     CLI     TCD     TXS     XBA
CLC     SEC     TCS     TXY     XCE
INX     INY     TDC     TXA
DEX     DEY     TSC     TYA
```

This list includes every instruction that explicitly sets or clears specific status flags.

Table 2–1: The Assembler Formats of the 65816 Addressing Modes

Name of Addressing Mode	*Assembler Format*
Implied	—
Accumulator	A
Immediate	#immediate
Program Counter Relative	rel
*Program Counter Relative Long	longrel
Stack	—
*Stack Relative	sr,S
*Stack Relative Indirect Indexed with Y	(sr,S),Y
*Block Move	srcbnk,destbnk
Absolute	addr
Absolute Indirect	(addr)
Absolute Indexed with X	addr,X
Absolute Indexed with Y	addr,Y
Absolute Indexed Indirect	(addr,X)
*Absolute Long	long
*Absolute Long Indexed with X	long,X
*Absolute Indirect Long	[addr]
Direct Page	dp
Direct Page Indirect	(dp)
Direct Page Indexed with X	dp,X
Direct Page Indexed with Y	dp,Y
Direct Page Indirect Indexed with Y	(dp),Y

Table 2–1: Continued

Name of Addressing Mode	*Assembler Format*
Direct Page Indexed Indirect with X	(dp,X)
*Direct Page Indirect Long	[dp]
*Direct Page Indirect Long Indexed with Y	[dp],Y

A = accumulator
addr = 16-bit address
destbnk = bank number
dp = direct page address
long = 24-bit address

longrel = 16-bit offset address
number = 8- or 16-bit constant
rel = 8-bit offset address
sr = stack pointer offset
srcbnk = bank number

NOTE: Addressing modes not supported by the 65C02 are marked with asterisks.

Accumulator A

The accumulator addressing mode is used by those instructions that manipulate the contents of the accumulator but that do not need to be supplied with an address or data. Here are some examples:

```
LSR     ROL
ASL     ROR
INC     DEC
```

To select this addressing mode, put the letter A in the operand field.

Immediate #immediate

The immediate addressing mode is used when the operand is a numeric constant. The assembler format for this mode is #immediate. That is, precede the number (or a symbol for a number) with the # symbol.

Here are some examples:

```
LDA #23          ;Decimal number
LDX #$AEF2       ;Hex numbers begin with $
CMP #%00000110   ;Binary numbers begin with %
REP #$20
ADC #256
```

Do not forget to include the # symbol when using this mode. If you do forget, the assembler thinks you are referring to an address and will deal with the number at that address, not the constant itself.

Program Counter Relative

This addressing mode is used by the 65816 branch-on-condition instructions: BCC, BCS, BEQ, BMI, BNE, BPL, BRA, BVC, and BVS. For this mode, the operand represents the offset to a target address, measured from the instruction following the branch instruction. This offset must be a signed value from -128 to +127. Transferring control to a target outside this range is possible only with unconditional JMP or BRL instructions.

When using this mode, it is not necessary for you to calculate the offset. Simply specify the address of the target location in the operand field of the instruction by referring to its label, as follows:

```
         BCC   MyTarget   ;Branch to labelled address
          .
          .
MyTarget NOP
```

The assembler calculates the offset for you when assembling the file.

Program Counter Relative Long

This addressing mode is used by the BRL and PER instructions only. For this mode, the operand is an offset from the start of the following instruction to a target address. The offset range is from -32768 to +32767, so you can access any location in the current bank. (Wraparound within a bank occurs if the offset is past the end of a bank.)

As with the program counter relative mode, you need only specify a target address when writing a program; the assembler will calculate the actual offset for you.

Stack

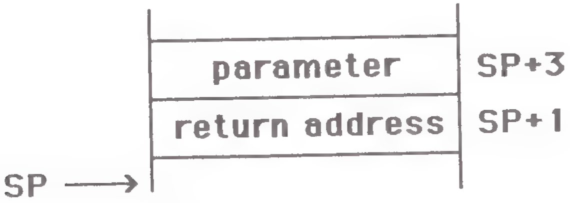
The stack addressing mode is really a variation of the implied mode, because the instruction mnemonic completely describes the operation. For push instructions, this mode causes data to be added to the top of the stack and the stack pointer to be decremented by the size of the data. For pull instructions, data is taken from the top of the stack and the stack pointer is incremented.

Stack Relative   sr,S

A common technique for passing parameters to a subroutine is to place the parameters on the stack prior to executing the JSR or JSL instruction. As the next chapter will cover, this is the technique you will use when calling GS tool set functions.

An easy way to retrieve parameters passed on the stack, or to return parameters in the stack, is to use the stack relative addressing mode. With this mode, you specify a one-byte number from $00 to $FF; the 65816 adds this number to the contents of the stack pointer to determine the effective address.

Suppose a program calls a subroutine by pushing one word of data on the stack before the JSR. When the subroutine gets control, the stack looks like this:



To access the parameter, use the following instruction:

```
    LDA      $03,S
```

The offset from the stack pointer is $03, because SP points to the address just below the JSR return address, and the return address is two bytes long. If the subroutine had been called with a JSL instruction, the offset would be $04 because a long return address is three bytes long.

Note that if the subroutine pushes any data on the stack after getting control, the offset to the parameter must be adjusted accordingly.

Stack Relative Indirect Indexed with Y   (sr,S),Y

This addressing mode is useful if the stack contains a pointer to a data structure in the current data bank. By specifying a stack offset from $00 to $FF and loading the Y register with an offset into the data structure, you can access any field in the data structure.

The assembler format for this mode is (sr,S),Y. The 65816 calculates the effective address by first adding sr to S. It then takes the pointer stored at that address and adds to it the value stored in the Y register.

One difficulty with this mode is that it does not work with long (24-bit) address pointers, and these are the types of pointers passed to GS tool set functions. In this situation, it is better to use the direct page indirect long indexed with Y addressing mode (see below). You can do this by defining a new direct page that begins at the base of the stack.

Block Move   srcbnk,destbnk

This addressing mode is used only by the MVN and MVP block move instructions. The form of the operand is two bank numbers separated by a comma; the first bank is the bank for the source block and the second is the bank for the destination block.

Absolute Addressing Modes

Absolute	`addr`
Absolute Long	`addr`
Direct Page	`dp`

The effective address for the absolute addressing mode is the 16-bit address defined by the label in the operand field. The 65816 forms the effective 24-bit address by combining this address with the number in the program bank register.

Here are some examples of the absolute addressing mode:

```
    LDA     MyData
    STX     TheResult
    ROL     BitFlags
```

In each case, the operand field contains a symbolic label for the address.

When using the absolute addressing mode, it is very important to ensure that the data bank register is the same as the program bank register, at least if the labels refer to offsets into the program bank (the usual case). The easiest way to do this is to execute PHK, PLB instructions at the beginning of the program.

If the data bank register is not set up properly, you must use the absolute long addressing mode. In this situation, the effective address is the 24-bit address specified in the operand; the data bank register is not referenced. Most assemblers, including Apple's APW assembler, default to absolute addressing if they cannot tell if the address is long (24-bit) or normal (16-bit). To force absolute long addressing, put a > symbol in front of the address.

If the operand is a location in direct page, you can use the direct page addressing mode instead of absolute or absolute long.

Absolute Indexed Addressing Modes

Absolute indexed with X	`addr,X`
Absolute indexed with Y	`addr,Y`
Absolute long indexed with X	`absaddr,X`
Direct page indexed with X	`dp,X`
Direct page indexed with Y	`dp,Y`

These indexed addressing modes are the same as the corresponding unindexed addressing modes, except that the 65816 adds the contents of an index register to the specified address to form the effective address. Here are some examples:

```
    LDA     TaskRecord,X      ;EA = TaskRecord+X
    STA     MyField,Y         ;EA = MyField+Y
    LDA     $E10000,X         ;EA = $E10000+X
    STA     $32,X             ;EA = $32+X
    LDX     $16,Y             ;EA = $16+Y
```

Use these modes to step into the data structure beginning at the base address. The base address is either a 16-bit address (addr), a 24-bit address (absaddr), or a direct page address (dp).

Indirect Addressing Modes

There are eight different indirect addressing modes you can use to access a data structure whose address is passed to you as a parameter:

Direct page indirect	`(dp)`
Direct page indirect long	`[dp]`
Absolute indirect	`(addr)`
Absolute indirect long	`[addr]`
Absolute indexed indirect	`(addr,X)`
Direct page indexed indirect with X	`(dp,X)`
Direct page indirect indexed with Y	`(dp),Y`
Direct page indirect long indexed with Y	`[dp],Y`

If you know the absolute position in memory of the data structure, you can use a direct addressing mode instead of an indirect addressing mode.

The indirect addressing modes all involve address pointers that are two (normal) or three (long) bytes in length. These pointers are stored in memory with the low-order byte or bytes first.

The indirect modes which use nondirect page addresses for storing the pointers work with JMP and JSR instructions only:

```
    JMP (addr)          ;16-bit pointer
    JMP [addr]          ;long pointer
    JMP (addr,X)        ;16-bit pointer
    JSR (addr,X)        ;16-bit pointer
```

For these modes, addr is the address at which the pointer to the data structure is stored. For (addr) and [addr] modes, this address must be in bank $00.

The (addr,X) mode is often used to pass control to a subroutine whose address is in a table of subroutine addresses beginning at addr. By putting the relative position of the subroutine address in X, you can call the subroutine with JSR (addr,X) or JMP (addr,X). That is because the 65816 forms the effective address by adding the value in the X register to the address specified in the operand.

For the other five indirect addressing modes, the pointer must be in a direct page location. The effective addresses are calculated as follows:

`(dp)` EA = the address stored at dp/dp+1
`[dp]` EA = the address stored at dp/dp+1/dp+2
`(dp),Y` EA = the address stored at dp/dp+1 plus Y

`[dp],Y`	EA = the address stored at dp/dp+1/dp+2 plus Y
`(dp,X)`	EA = the address stored at dp+X/dp+X+1

Note: The effective addresses are addresses in the bank stored in the data bank register.

Use the (dp) or [dp] or (dp,X) modes if you need to access only the first byte (8-bit register) or word (16-bit register) of the data structure. To manipulate other portions of a data structure, you must use (dp),Y or [dp],Y after setting the Y register to the appropriate value.

The [dp],Y addressing mode is useful for accessing data structures whose long pointers are passed on the stack to a subroutine. Here is an example of a subroutine that stores a result in bytes $20/$21 of a data structure whose address is pushed on the stack before the subroutine is called with JSL:

```
; 3 bytes at top of stack on entry (from JSL)
PHD                       ;Save d.p. register (adds 2 to SP)
TSC
TCD                       ;Make direct page = stack pointer
LDA       #$8000          ;This is the value to return
LDY       #20             ;access 20th byte
STA       [$06],Y         ;store the result ($06 = 2+3+1)
PLD                       ;Restore original direct page
RTL                       ;(Sub called with JSL)
```

Rather than remove the pointer from the stack and put it in the current direct page, this subroutine defines a new direct page that begins at the address stored in the stack pointer. Because this occurs after the PHD instruction, the direct page address of the stack parameter is $06; the long return address (pushed on the stack by JSL) is at $03-$05, and the copy of the direct page register is at $01-$02.

INTERRUPTS

An interrupt is an electronic signal that demands the immediate attention of the microprocessor. The signal can originate from either a hardware device or a programming instruction. The four basic types of hardware interrupts are:

- IRQ (interrupt request)
- NMI (nonmaskable interrupt)
- Reset
- Abort

The two types of software interrupts are:

- BRK (break)
- COP (co-processor)

This section takes a look at how the 65816 deals with interrupts.

The IRQ Hardware Interrupt

An IRQ hardware interrupt is an electrical signal from a peripheral device that activates the IRQ (interrupt request) pin on the 65816. When the 65816 senses an active IRQ signal, it stops executing the current program and switches to another program, called an interrupt handler, to deal with the source of the interrupt. When the handler ends, control returns to the original program and execution continues.

It is easier for a program to deal with input devices that interrupt the system because there is less risk of missing incoming data. If the device does not interrupt, the program must periodically check its status to see if the device has data ready. If the program does not check often enough (perhaps because it is executing a time-consuming subroutine), it will get behind and lose data.

Some of the sources of IRQ interrupts on the GS are the serial ports, the mouse, the keyboard, the clock, the video controller, and the Ensoniq DOC. The serial ports, for example, can be programmed to generate interrupts when they receive data. The interrupt handler will then read the data and store it in a buffer so that it could be retrieved by the program when it is ready for more input.

If you set the interrupt disable flag in the 65816 status register, the 65816 will ignore IRQ interrupts. You may want to do this if your program is in the middle of a critical timing loop or if the program is not re-entrant (capable of being called while it is in use) and there is a possibility the interrupt handler may call the program recursively. In general, interrupts should be left on at all other times.

When an IRQ interrupt occurs, the 65816 takes one of two actions, depending on its current mode. In native mode, it pushes the program bank register, the program counter, and the status register on the stack. In emulation mode, only the program counter and the status register are pushed. Then the 65816 clears the decimal mode flag in the status register and sets the interrupt disable flag so that nothing will interrupt the interrupt-handling subroutine. Finally, the 65816 passes control to a vector in bank $00 associated with IRQ interrupts. As shown in table 2–2, the position of this vector depends on the operating mode. The interrupt handler ultimately passes control to a user vector at $3FE/$3FF in bank $00—that is, if control is not handled internally.

An interrupt handler must end with an RTI instruction because RTI pops the values pushed on the stack when interrupt processing begins. Because only the

Table 2–2: 65816 Interrupt Vectors

Emulation Mode	*Vectors*	*Native Mode*	*Vectors*
$00FFFE/FF	IRQ, BRK	$00FFEE/EF	IRQ
$00FFFC/FD	Reset	$00FFEA/EB	NMI
$00FFFA/FB	NMI	$00FFE8/E9	Abort
$00FFF8/F9	Abort	$00FFE6/E7	BRK
$00FFF4/F5	COP	$00FFE4/E5	COP
		$00FFFC/FD	Reset

program counter and the status register are saved on the stack, the interrupt handler must preserve the states of the other 65816 registers it alters.

The preferred way to install interrupt handlers in a ProDOS environment is to use the ProDOS ALLOC—INTERRUPT command. (See chapter 10.) In these cases the interrupt handler does not have to preserve registers and can end with an RTS (ProDOS 8) or RTL (ProDOS 16) instruction. This is because the internal ProDOS interrupt dispatcher takes care of saving registers and returning control properly.

Other Hardware Interrupts

Devices can also generate NMIs (nonmaskable interrupts). The 65816 handles them just like IRQs except that it uses a different interrupt vector and it never ignores them even if the interrupt disable flag is set. A peripheral might cause an NMI if it has to take over the system in an emergency situation, such as an impending loss of power. The NMI interrupt handler in the GS eventually passes control to a user-installed subroutine whose address is stored at $3FC/$3FD in bank $00.

A Reset interrupt occurs when you turn the GS on or when you press Control-Reset. When Reset is caused by the power turning on, the system boots the start-up disk. When power is turned on, Resets are handled by eventually passing control to the subroutine whose address is stored at $3F2/$3F3 in bank $00, but only if the byte at $3F4 is equal to the number generated by exclusive-ORing the number at $3F3 with the constant $A5. If it is not, the system is rebooted instead.

The last type of hardware interrupt is the Abort interrupt. It is supposed to be generated by a hardware memory-management unit when it detects a program trying to access memory that is off-limits to it. On the GS, only cards installed in the memory expansion slot can generate Abort interrupts.

The addresses of the vectors for NMI, Reset, and Abort interrupts are shown in table 2–2.

Software Interrupts

You can also generate interrupt signals by executing a BRK or COP instruction inside a program. The 65816 reacts to such interrupt signals just as it reacts to an active IRQ signal, but it uses a different interrupt vector. Neither the BRK nor the COP instruction can be masked by the interrupt disable flag in the status register.

BRK and COP have separate interrupt vectors but are handled in much the same way by the GS monitor. (The COP instruction was designed to pass control to a 65816 coprocessor chip, but such a chip is not available yet.) When you start up ProDOS 8, however, both vectors eventually pass control to the same RAM vector at $3F0/$3F1 in bank $00. This vector points to a monitor subroutine that displays the contents of the 65816 registers.

BRK and COP are both two-byte instructions even though the second byte has no particular meaning. The BRK or COP handler you install can examine the second byte and interpret it any way it wishes. It is essentially a parameter that is passed to the handler.

Note that in emulation mode, BRK and IRQ interrupts share the same interrupt vector. To distinguish between them, the system examines the break bit in the status register; if it is 1, the interrupt was caused by a BRK instruction and the GS monitor passes control to the vector stored at $3F0/$3F1 instead of to that at $3FE/$3FF.

CREATING PROGRAMS WITH THE APPLE IIGS PROGRAMMER'S WORKSHOP

The Apple IIGS Programmer's Workshop (APW) contains all the programs and support files you will need to develop 65816 assembly language programs on the GS. There are four key modules:

- The shell (command interpreter)
- The text editor
- The linker
- The 65816 assembler

By acquiring APW-compatible compilers, you can also use APW to develop programs written in higher-level languages like C and Pascal.

To begin writing a 65816 program, start up APW and ensure that the active language is 65816 assembly by entering the ASM65816 command from the shell. (This language is the initial default, so you can skip this step if you have not switched to another language.) You must choose the correct language because the assembler will not deal with source files created by the editor when another language is active.

The next step is to invoke the editor with the command:

```
EDIT pathname
```

where "pathname" represents the name of an existing source file or the name of the new file you want to create.

Now the hard part: you must write your application! To do this requires an intimate knowledge of the syntactical requirements of the assembler.

Source Code Format

Lines of assembly language source code are made up of four fields, separated by one or more spaces, in the following order:

- Label field (usually optional)
- Instruction field
- Operand field
- Comment field (optional)

The major exceptions are comment lines, which begin with ";", "*", or "!" symbols. You can put whatever you like in these lines because they are ignored by the assembler.

The Label Field. The label field optionally contains a symbolic label, usually for marking a position within the program to which other parts of the program can refer. Labels can also be associated with numeric constants with the EQU and GEQU assembler directives (see below).

A label must begin with an alphabetic character (A to Z) or an underscore (__). Subsequent characters can be alphabetic (A to Z), numeric (0 to 9), the tilde (˜), or the underscore (__). The maximum length is 255 characters. The assembler usually ignores the case of alphabetic characters, but you can make it case-sensitive with the CASE ON directive.

The Instruction Field. The instruction field contains the 65816 instruction or an assembler directive. Directives are commands to the assembler, not 65816 instructions. They tell the assembler to do such things as set or clear internal status flags, define symbols, allocate data storage areas, save object code to disk, and create macros.

If the label field contains an entry, the instruction field must not be blank; if no instruction is wanted, use the ANOP (assembler no operation) directive. Unlike some assemblers, the 65816 assembler does not allow you to define lines that contain only a label.

The Operand Field. The operand field contains the addressing mode for the instruction or a parameter for the assembler directive. It may be blank if the instruction or directive requires no explicit operand.

The Comment Field. The comment field, which begins with a ; symbol, can contain any explanatory text you like. Use it to document your program code.

The source code shell for a typical 65816 program looks something like this:

```
           KEEP  MyProgram      ;Name of object code file
           MCOPY Macros.Mac     ;Name of macro file
CodeSeg1   START                ;Start of main code segment
           USING GlobalData     ;It uses GlobalData data
Private    NOP                  ;"Private" is a local label
           <put program code
           here>
  MySub    ENTRY                ;A global entry point
           <more program code
           here>
           END
CodeSeg2   START                ;Another code segment
           USING GlobalData     ;It can use GlobalData too!
           JSR CodeSeg1         ;You can access through
           JSR MySub            ;global entry points.
           JSR Private          ;Error!!! But not local ones!
           <code for another
           segment here>
           END
           COPY OtherCode.Asm   ;Name of other file to include
GlobalData DATA
           <put data allocation
           statements here>
           END
```

The "KEEP pathname" directive saves to disk the object code generated by the assembler under names of the form "pathname.ROOT" (for the first code segment in the program) and "pathname.A" (for all other segments, if any). The ProDOS file type code for these files is $B1 and the standard mnemonic is OBJ. The KEEP directive must appear before the first START directive.

"MCOPY pathname" tells the assembler the name of the file in which macros referred to in the source file are defined. Macros will be described in greater detail below.

"COPY pathname" tells the assembler to load the specified source file and assemble its statements. On completion, assembly continues with the next line in the main file. Use it to include standard source code fragments in your main source program file.

All 65816 instructions in a program source file must fall inside one or more code segments defined by START and END directives. START marks the beginning of a code segment and must be associated with a label that identifies the subroutine or main program in the segment. END marks the end of the segment.

A single source file may contain any number of code segments. The advantage of partitioning a program into several segments is that all labels defined inside the segment are local to the segment. This means that the same labels can be reused in another segment without generating an assembly error. This is especially convenient if you are developing extremely large programs in which the risk of symbol duplication is great.

The label used with the START directive is global, which means that it cannot be used as a label anywhere else in the program. It identifies the primary entry point to the subroutine contained in the segment. You can use the ENTRY directive to define other global entry points into the subroutine, if you wish. Like START, ENTRY requires a label.

You can define program segments that contain only data (no instructions) with the DATA directive. A data segment must end with the END directive.

As you might expect, the labels inside a data segment are local to that segment, although the name of the data segment is global. You can, however, associate a data segment with a code segment so that the data segment's labels, in effect, become local to the code segment as well. To do this, use the "USING DataSegName" directive. A data segment may be used by any code segment in this way.

Local and Global Labels

You will probably want to associate symbolic labels to numeric constants to make your source code easier to understand and to debug. You can do this with the EQU (equate) or GEQU (global equate) directives.

Here are some examples:

```
True     GEQU     $8000            ;takes effect throughout program
BitPos   EQU      3                ;takes effect in current segment
```

GEQU creates a global constant—one that takes effect in every code and data segment in the source file. The same label may not be redefined elsewhere in the program.

EQU creates a local constant, which takes effect only inside the segment in which the constant was defined. EQU labels can be redefined in other segments. If an EQU label has the same name as a GEQU label, the EQU label takes precedence.

ProDOS 16 Entry Conditions

Just before ProDOS 16 passes control to an application, it puts the system into a standard operating state:

- full native mode (*m*=0, *x*=0, *e*=0)
- the X and Y registers are zero

- the A register contains the ID tag for the program
- shadowing of the text screen is on
- shadowing of the graphics area is off
- I/O shadowing and language card operation is on

(The concepts of ID tags and shadowing will be explained in chapter 4.)

In addition, ProDOS 16 reserves a space in bank $00 for use as a direct page and stack area. This is normally a 1K space, but can be changed using advanced assembly/linking techniques. On entry to the program, the direct page register contains the address of the first byte in this space and the stack pointer points to the last byte in the space.

Mode Considerations

After ProDOS 16 loads an executable application into memory, it puts the 65816 into full native mode with 16-bit A, X, and Y registers; it then passes control to the beginning of the first code segment. This means the program does not have to change modes with XCE or REP if it needs to run in full native mode.

The assembler must be told of any changes in the size of the accumulator and index registers so that it can assemble the following instructions correctly:

```
LDA #immediate   SBC #immediate
LDX #immediate   EOR #immediate
LDY #immediate   BIT #immediate
ADC #immediate   ORA #immediate
AND #immediate
```

The assembler initially assumes settings of LONGA ON (16-bit accumulator) and LONGI ON (16-bit index registers). LONGA and LONGI are directives that appear in the instruction field of a line of source code. If the directive is set to OFF, the assembler considers the corresponding register to be 8 bits in size.

Any time you change register sizes using REP or SEP, you must also tell the assembler by adjusting LONGA and LONGI to the appropriate values.

Numbering Systems

The APW assembler understands many different numbering systems. The ones you will use most often are decimal, hexadecimal, and binary. To indicate which system you are using, place a special identification character in front of the number:

$ Hexadecimal number (for example, $FFAD)
% Binary number (for example, %01100111)

No special character is needed for a decimal number, because decimal is the default numbering system.

Isolating the Words in a Long Address

You often need to load a register with the low- or high-order word of a particular address. Loading the low-order portion is easy:

```
    LDA      #Address        ;Load address (low)
```

Because the A register (or X, Y) is 16 bits in size, the upper 8 bits of Address are ignored. To load the high-order portion, you must append "|-16" to the address operand (which means "shift right by 16 bits") or precede it with "^" as shown in the following examples:

```
    LDA      #Address|-16    ;Load high-order word
    LDA      #^Address       ;(alternative method)
```

In both cases, the bank portion of the 24-bit address is the low-order 8 bits of the accumulator. The top 8 bits are zero.

For PEA operations, do not include the "#" because the addressing mode is absolute, not immediate. To push the two words of an address on the stack, do this:

```
    PEA       Address|-16    ;Push address high
    PEA       Address        ;Push address low
```

(The APW assembler does not allow an ^Address operand for PEA instructions.) This is the preferred technique for pushing address pointers because it does not require using a 65816 register.

Forcing Addressing Modes

When you specify an address as an operand, the APW assembler normally considers it to be an absolute address, as opposed to a long absolute or direct page address, and assembles the instruction accordingly. Exceptions are symbolic addresses defined by EQU or GEQU directives; here, APW is able to determine the appropriate addressing mode.

Using absolute addresses may or may not be appropriate, depending on the state of the data bank register. If the symbol for the address identifies a location inside the program itself, absolute addressing is improper if the data bank register is not the same as the program bank register. In this situation you can either make the two banks the same (with PHK, PLB instructions) or force absolute long addressing by preceding the name of the address label with the > symbol.

For example, to load the A register with whatever is stored at MyAddress, execute the following code:

```
    LDA     )MyLabel          ;Use absolute long addressing
```

This forces the 65816 to read the value from the program bank as intended.

Similarly, you can force direct page addressing with the "<" symbol and absolute addressing with the "|" symbol:

```
    LDA     <MyAddress        ;Force direct page addressing
    LDA     |MyAddress        ;Force absolute addressing
```

You would use these only if the assembler would normally assemble these instructions using a different addressing mode.

Data Allocation Directives

You will use data allocation directives to reserve space for data inside the program, or for storing specific values inside the program. To reserve space, use the DS directive:

```
    DS      45      ;Reserve 45 bytes
```

The operand of the directive is the number of bytes to reserve. The line containing a DS directive may contain a label.

To store specific values in the object code, use the DC (define constant) directive instead of DS. It is of the form:

```
    label         DC       constant_def
```

where constant_def defines the type of data to be stored. Here are examples of the most common constant definitions:

I1‘22’	One byte
I2’396’	Two bytes (one word)
I’298’	Two bytes (one word)
I4’$8000’	Four bytes (long word)
H’FE128A’	Hexadecimal digits
B’100110’	Binary digits
C’Text String’	Sequence of characters

The I*x* definition (*x* is an integer from 1 to 8), stores multiple bytes in memory, low-order bytes first. This is the order expected by 65816 instructions. If *x* is omitted, two-byte words are stored. The number specified can be in binary, decimal, or hexadecimal form.

The H definition stores successive pairs of hexadecimal digits in successive memory bytes. (Any spaces in the digit string are ignored.) If an odd number of digits is specified, the low-order four bits of the last byte will be set to 0.

The B definition stores successive groups of eight binary digits in successive memory bytes, ignoring any spaces. If the number of digits specified is not an even multiple of eight, the last byte is padded on the right with 0 bits.

The C definition stores the ASCII codes for the specified character string in memory. The MSB ON and MSB OFF directives control the setting of the high bit for these string constants. The default is MSB OFF, meaning that bit 7 of each byte is 0. Use MSB ON if you wish to force bit 7 to 1.

Several constant definitions can be included on one line by separating them with commas. For example, the command:

```
    DC   I1'0',I2'$FDDA',H'AE'
```

stores the bytes $00 $DA $FD $AE in the object code at assembly time.

To define several items of the same general type, put the items, separated by commas, inside the single quote marks of the constant definition string:

```
    DC   I2'34,$A321,16'   ;Stored as 22 00 21 A3 10 00
```

Another nice feature of the DC directive is that it gives you the ability to put a repeat count in front of a constant definition. For example, to generate six copies of $FDED, use the following command:

```
    DC   6I'$FDED'
```

Notice that in this example the "2" after the "I" was omitted. As indicated above, the assembler assumes it is dealing with words in this situation.

Listing Directives

The APW assembler supports several directives that affect the output of the assembly process:

- LIST (display object code)
- SYMBOL (display symbol table)
- ABSADDR (display code addresses)

- INSTIME (display instruction cycle times)
- GEN (display macro expansions)

These directives can either be ON or OFF; put the ON/OFF status indicator in the operand field. The defaults for all these directives is OFF.

Creating the Macro File

A macro is an assembler command that expands into a series of 65816 instructions (or assembler directives) during the assembly process. By assigning commonly used short code fragments to a macro command, you can develop programs much more rapidly and make them much more readable at the same time.

As the next chapter will discuss, the GS supports many tool set functions that you invoke by loading the X register with an ID number and then doing a JSL to $E10000. You do not have to memorize these ID numbers, because Apple provides with APW a set of macro definitions that associates easy-to-remember names with each of the LDX/JSL function calls.

You can define powerful macros for use with the APW assembler. For a comprehensive description of how to create macros, see the *APW Assembler Reference*. To get a feel for how to create macros, look at this one, called PushAddr, which pushes the address of a program or data area on the stack:

```
         MACRO                      ;Start macro definition
&lab     PushAddr &addr             ;Name of macro, parameters
&lab     PEA      &addr|-16         ;Body of macro
         PEA      &addr             ;Body of macro
         MEND                       ;End of macro
```

To invoke this macro, you would place a line such as:

```
MyLabel PushAddr MyRecord
```

in your program source code. When the assembler expands the macro into 65816 instructions, it replaces &lab with MyLabel and &addr with MyRecord wherever they appear in the body of the macro definition. For the above example, this means the following statements would be generated:

```
MyLabel PEA      MyRecord|-16
        PEA      MyRecord
```

The MACRO directive marks the beginning of the macro definition. The next line contains the name of the macro in the instruction field. It also defines the parameters you can pass to the macro; their names begin with the & symbol. For PushAddr, one parameter is a label appearing in the label field (&lab); the other is an address appearing in the operand field (&addr). Multiple nonlabel parameters appear in the operand field of a macro, separated by commas. The MEND directive marks the end of the macro definition.

With conditional assembly techniques you can define macros whose behavior depends on many things, such as whether the parameters you specify are immediate values or addresses. Examples of macros like this are given in listing 2–1. (These macros are ideal for passing parameters to tool set functions and for retrieving results, as will be shown in the next chapter.) Consult the *APW Assembler Reference* for a detailed explanation of how to use these advanced macro features.

Almost every program you write for the GS will use macros, primarily the ones that assign symbolic names to tool set functions. The macros must appear in a separate file that is referred to by an "MCOPY pathname" directive in the primary source code file.

APW comes with a program called MACGEN that you can use to extract the macros needed by a program from the standard APW macro files and to put them in a custom macro file. For example, suppose your source program is called WINDOWS.ASM and it uses macros defined in M16.QUICKDRAW and M16.WINDOW. To place all the macros used by the program in a file called WINDOWS.MAC (the file named in the program's MCOPY command), type in this command:

```
MACGEN WINDOWS.ASM WINDOWS.MAC M16.QUICKDRAW M16.WINDOW
```

MACGEN scans the source file (the first file named) for macro references and places their definitions in the macro file (the second file named). It searches the third and fourth files (and any more you might specify on the command line) for these macro definitions. If not all macros are found, MACGEN asks you to enter the name of another macro file to search; it keeps asking until it finds every definition.

Quite often you will create a macro file, then change the program slightly so that it uses a few new macros. To add just the new macros to the existing program macro file, specify the program macro file as one of the MACGEN search files:

```
MACGEN WINDOWS.ASM WINDOWS.MAC WINDOWS.MAC M16.WINDOW
```

By doing this, you avoid having to specify the names of all the system macro files containing the macros you already have in the program macro file.

Creating an Application with APW

The APW assembler processes a 65816 source code file and converts it into an object code file. The resultant object code file is not the executable program, however. Instead, it contains code modules that the APW linker must process first. It is the linker that creates the executable ProDOS 16 load file. A load file contains the program code and a header that tells the System Loader (the tool set responsible for loading programs into memory) how to make the program run properly at the load address.

Object code files produced by the assembler are in a special language-independent format defined by Apple. High-level language compilers that work in the APW environment (notably C and Pascal) use this same format. The linker does not care which language created the object code, so you can develop portions of a program in different languages, then combine them with the linker to create the final application.

The APW command for assembling and linking simple applications in one step is ASML. If you need to perform special linking operations (such as including code libraries), you have to separately process a linker control file written in the LINKED language instead; for information on the LINKED language, see the *APW Assembler Reference*.

After assembling and linking, APW creates an executable ProDOS 16 load file with a file type code of $B5 (the standard mnemonic is EXE). This program can be run from the APW shell simply by typing in its name. When it ends (with an RTL instruction or a ProDOS 16 QUIT command), control returns to the shell.

Once you have thoroughly debugged the program, you should change its file type code to $B3 (the mnemonic is S16) so that ProDOS 16 will recognize it as an application. Do this with the APW FILETYPE command:

```
FILETYPE pathname $B3
```

where $B3 is the new file type code. You can specify the S16 mnemonic instead of the hexadecimal code, if you wish.

An S16 program can be executed from the APW shell, but it will crash the system if it ends with an RTL instruction. This is because the shell removes itself from the system when it passes control to the program, so it will not be there on return. (With EXE programs the shell stays intact.) The moral is to always end a program with a ProDOS QUIT command. This command is described in chapter 10.

REFERENCE SECTION

Table R2–1: The Complete 65816 Instruction Set

					Effect on Flags
	Instruction		*Addressing*		NV-BDIZC *e*=1
Mnemonic	*Operation*	*Opcode*	*Mode*	*Cycles*	NVmxDIZC *e*=0
ADC	add to accumulator with carry	$69	#immediate	2[1]	NV....ZC
		$6D	addr	4[1]	
		$6F	long	5[1]	
		$65	dp	3[1]	

					Effect on Flags
	Instruction		*Addressing*		NV-BDIZC *e*=1
Mnemonic	*Operation*	*Opcode*	*Mode*	*Cycles*	NVmxDIZC *e*=0
		$71	(dp),Y	5[1,3]	
		$77	[dp],Y	6[1]	
		$61	(dp,X)	6[1]	
		$75	dp,X	4[1]	
		$7D	addr,X	4[1,3]	
		$7F	long,X	5[1]	
		$79	addr,Y	4[1,3]	
		$72	(dp)	5[1]	
		$67	[dp]	6[1]	
		$63	sr,S	4[1]	
		$73	(sr,S),Y	7[1]	
AND	logical AND with accumulator	$29	#immediate	2[1]	N.....Z.
		$2D	addr	4[1]	
		$2F	long	5[1]	
		$25	dp	3[1]	
		$31	(dp),Y	5[1,3]	
		$37	[dp],Y	6[1]	
		$21	(dp,X)	6[1]	
		$35	dp,X	4[1]	
		$3D	addr,X	4[1,3]	
		$3F	long,X	5[1]	
		$39	addr,Y	4[1,3]	
		$32	(dp)	5[1]	
		$27	[dp]	6[1]	

Mnemonic	Instruction Operation	Opcode	Addressing Mode	Cycles	Effect on Flags NV-BDIZC *e*=1 NVmxDIZC *e*=0
		$23	sr,S	4[1]	
		$33	(sr,S),Y	7[1]	
ASL	arithmetic shift left	$0E	addr	6[2]	N.....ZC
		$06	dp	5[2]	
		$0A	A	2	
		$16	dp,X	6[2]	
		$1E	addr,X	7[2]	
BCC	branch on carry clear (C=0)	$90	rel	2[4,5]	....
BCS	branch on carry set (C=1)	$B0	rel	2[4,5]	....
BEQ	branch on equal (Z=1)	$F0	rel	2[4,5]	....
BIT	Logically AND the accumulator with the operand; transfer high-order bits to N and V (but not if #immediate)	$89	#immediate	2[1]	NV....Z.
		$2C	addr	4[1]	
		$24	dp	3[1]	
		$34	dp,X	4[1]	
		$3C	addr,X	4[1,3]	
BMI	branch on minus (N=1)	$30	rel	2[4,5]	....
BNE	branch on not equal (Z=0)	$D0	rel	2[4,5]	....
BPL	branch on plus (N=0)	$10	rel	2[4,5]	....

					Effect on Flags
	Instruction		*Addressing*		NV-BDIZC *e*=1
Mnemonic	*Operation*	*Opcode*	*Mode*	*Cycles*	NVmxDIZC *e*=0
BRA	branch relative always	$80	rel	3^5	....
BRK[a]	break interrupt	$00	{stack}	7^6	....01..
BRL	branch relative long	$82	longrel	4	....
BVC	branch on overflow clear (V=0)	$50	rel	$2^{4,5}$	....
BVS	branch on overflow set (V=1)	$70	rel	$2^{4,5}$	....
CLC	clear carry flag	$18	{implied}	2	0
CLD	clear decimal flag	$D8	{implied}	2	....0...
CLI	clear IRQ disable flag	$58	{implied}	2	.....0..
CLV	clear overflow flag	$B8	{implied}	2	.0......
CMP	compare with accumulator	$C9	#immediate	2^1	N.....ZC
		$CD	addr	4^1	
		$CF	long	5^1	
		$C5	dp	3^1	
		$D1	(dp),Y	$5^{1,3}$	
		$D7	[dp],Y	6^1	
		$C1	(dp,X)	6^1	
		$D5	dp,X	4^1	
		$DD	addr,X	$4^{1,3}$	
		$DF	long,X	5^1	

Mnemonic	*Instruction Operation*	*Opcode*	*Addressing Mode*	*Cycles*	*Effect on Flags* NV-BDIZC *e*=1 NVmxDIZC *e*=0
		$D9	addr,Y	4[1,3]	
		$D2	(dp)	5[1]	
		$C7	[dp]	6[1]	
		$C3	sr,S	4[1]	
		$D3	(sr,S),Y	7[1]	
COP	co-processor interrupt	$02	{stack}	7[6]	....01..
CPX	compare with X	$E0	#immediate	2[7]	N.....ZC
		$EC	addr	3[7]	
		$E4	dp	2[7]	
CPY	compare with Y	$C0	#immediate	2[7]	N.....ZC
		$CC	addr	4[7]	
		$C4	dp	3[7]	
DEC	decrement	$CE	addr	6[2]	N.....Z.
		$C6	dp	5[2]	
		$3A	A	2	
		$D6	dp,X	6[2]	
		$DE	addr,X	7[2]	
DEX	decrement X	$CA	{implied}	2	N.....Z.
DEY	decrement Y	$88	{implied}	2	N.....Z.
EOR	exclusive OR with accumulator	$49	#immediate	2[1]	N.....Z.
		$4D	addr	4[1]	
		$4F	long	5[1]	

Mnemonic	*Instruction Operation*	*Opcode*	*Addressing Mode*	*Cycles*	*Effect on Flags* NV-BDIZC *e*=1 NVmxDIZC *e*=0
		$45	dp	3^{1}	
		$51	(dp),Y	$5^{1,3}$	
		$57	[dp],Y	6^{1}	
		$41	(dp,X)	6^{1}	
		$55	dp,X	4^{1}	
		$5D	addr,X	$4^{1,3}$	
		$5F	long,X	5^{1}	
		$59	addr,Y	$4^{1,3}$	
		$52	(dp)	5^{1}	
		$47	[dp]	6^{1}	
		$43	sr,S	4^{1}	
		$53	(sr,S),Y	7^{1}	
INC	increment	$EE	addr	6^{2}	N.....Z.
		$E6	dp	5^{2}	
		$1A	A	2	
		$F6	dp,X	6^{2}	
		$FE	addr,X	7^{2}	
INX	increment X	$E8	{implied}	2	N.....Z.
INY	increment Y	$C8	{implied}	2	N.....Z.
JMP	jump	$4C	addr	3	....
		$6C	(addr)	5	
		$7C	(addr,X)	6	

Mnemonic	*Instruction Operation*	*Opcode*	*Addressing Mode*	*Cycles*	*Effect on Flags* NV-BDIZC *e*=1 NVmxDIZC *e*=0
JML	jump long	$5C	long	4	....
		$DC	[addr]	6	....
JSL	jump to long subroutine	$22	long	8	....
JSR	jump to subroutine	$20	addr	6	....
		$FC	(addr,X)	8	
LDA	load the accumulator	$A9	#immediate	2^{1}	N.....Z.
		$AD	addr	4^{1}	
		$AF	long	5^{1}	
		$A5	dp	3^{1}	
		$B1	(dp),Y	$5^{1,3}$	
		$B7	[dp],Y	6^{1}	
		$A1	(dp,X)	6^{1}	
		$B5	dp,X	5^{1}	
		$BD	addr,X	$5^{1,3}$	
		$BF	long,X	6^{1}	
		$B9	addr,Y	$4^{1,3}$	
		$B2	(dp)	5^{1}	
		$A7	[dp]	6^{1}	
		$A3	sr,S	4^{1}	
		$B3	(sr,S),Y	7^{1}	
LDX	load the X register	$A2	#immediate	2^{7}	N.....Z.
		$AE	addr	4^{7}	
		$A6	dp	3^{7}	

Mnemonic	*Instruction* *Operation*	*Opcode*	*Addressing* *Mode*	*Cycles*	*Effect on Flags* NV-BDIZC *e*=1 NVmxDIZC *e*=0
		$B6	dp,Y	4[7]	
		$BE	addr,Y	4[7,3]	
LDY	load the Y register	$A0	#immediate	2[7]	N.....Z.
		$AC	addr	4[7]	
		$A4	dp	3[7]	
		$B4	dp,X	4[7]	
		$BC	addr,X	4[7,3]	
LSR	logical shift right	$4E	addr	6[2]	0.....ZC
		$46	dp	5[2]	
		$4A	A	2	
		$56	dp,X	6[2]	
		$5E	addr,X	7[2]	
MVN	move block backward	$54	src,dest	7[8]	....
MVP	move block forward	$44	src,dest	7[8]	....
NOP	no operation	$EA	{implied}	2	....
ORA	logical OR with accumulator	$09	#immediate	2[1]	N.....Z.
		$0D	addr	4[1]	
		$0F	long	5[1]	
		$05	dp	3[1]	
		$11	(dp),Y	5[1,3]	
		$17	[dp],Y	6[1]	
		$01	(dp,X)	6[1]	

Mnemonic	Instruction Operation	Opcode	Addressing Mode	Cycles	*Effect on Flags* NV-BDIZC *e*=1 NVmxDIZC *e*=0
		$15	dp,X	4^{1}	
		$1D	addr,X	$4^{1,3}$	
		$1F	long,X	5^{1}	
		$19	addr,Y	$4^{1,3}$	
		$12	(dp)	5^{1}	
		$07	[dp]	6^{1}	
		$03	sr,S	4^{1}	
		$13	(sr,S),Y	7^{1}	
PEA	push effective address	$F4	addr	5	. . . .
PEI	push effective address indirect	$D4	(dp)	6	. . . .
PER	push effective address relative	$62	rel	6	. . . .
PHA	push the accumulator	$48	{stack}	3^{1}	. . . .
PHB	push data bank register	$8B	{stack}	3	. . . .
PHD	push direct page register	$0B	{stack}	4	. . . .
PHK	push program bank register	$4B	{stack}	3	. . . .
PHP	push status register	$08	{stack}	3	. . . .
PHX	push the X register	$DA	{stack}	3^{7}	. . . .

					Effect on Flags
	Instruction		*Addressing*		NV-BDIZC *e*=1
Mnemonic	*Operation*	*Opcode*	*Mode*	*Cycles*	NVmxDIZC *e*=0
PHY	push the Y register	$5A	{stack}	3[7]	....
PLA	pull the accumulator	$68	{stack}	4[1]	N.....Z.
PLB	pull data bank register	$AB	{stack}	4	N.....Z.
PLD	pull direct page register	$2B	{stack}	5	N.....Z.
PLP	pull status register	$28	{stack}	4	NVmxDIZC
PLX	pull the X register	$FA	{stack}	4[7]	N.....Z.
PLY	pull the Y register	$7A	{stack}	4[7]	N.....Z.
REP	reset status bits	$C2	#immediate	3	NVmxDIZC
ROL	rotate left	$2E	addr	6[2]	N.....ZC
		$26	dp	5[2]	
		$2A	A	2	
		$36	dp,X	6[2]	
		$3E	addr,X	7[2]	
ROR	rotate right	$6E	addr	6[2]	N.....ZC
		$66	dp	5[2]	
		$6A	A	2	
		$76	dp,X	6[2]	
		$7E	addr,X	7[2]	
RTI	return from interrupt	$40	{stack}	6[6]	NVmxDIZC

	Instruction		Addressing		Effect on Flags NV-BDIZC *e*=1
Mnemonic	*Operation*	*Opcode*	*Mode*	*Cycles*	NVmxDIZC *e*=0
RTL	return from subroutine long	$6B	{stack}	6	. . . .
RTS	return from subroutine	$60	{stack}	6	. . . .
SBC	subtract from accumulator with carry	$E9	#immediate	2[1]	NV. . . . ZC
		$ED	addr	4[1]	
		$EF	long	5[1]	
		$E5	dp	3[1]	
		$F1	(dp),Y	5[1,3]	
		$F7	[dp],Y	6[1]	
		$E1	(dp,X)	6[1]	
		$F5	dp,X	4[1]	
		$FD	addr,X	4[1,3]	
		$FF	long,X	5[1]	
		$F9	addr,Y	4[1,3]	
		$F2	(dp)	5[1]	
		$E7	[dp]	6[1]	
		$E3	sr,S	4[1]	
		$F3	(sr,S),Y	7[1]	
SEC	set carry flag	$38	{implied}	2	 1
SED	set decimal flag	$F8	{implied}	2	. . . . . 1 . . .
SEI	set IRQ disable flag	$78	{implied}	2	. . . . . . 1 . .

					Effect on Flags
	Instruction		*Addressing*		NV-BDIZC *e*=1
Mnemonic	*Operation*	*Opcode*	*Mode*	*Cycles*	NVmxDIZC *e*=0
SEP	set status bits	$E2	#immediate	3	NVmxDIZC
STA	store the accumulator	$8D	addr	4[1]	. . . .
		$8F	long	5[1]	
		$85	dp	3[1]	
		$91	(dp),Y	6[1,3]	
		$97	[dp],Y	6[1]	
		$81	(dp,X)	6[1]	
		$95	dp,X	4[1]	
		$9D	addr,X	5[1,3]	
		$9F	long,X	5[1]	
		$99	addr,Y	5[1,3]	
		$92	(dp)	5[1]	
		$87	[dp]	6[1]	
		$83	sr,S	4[1]	
		$93	(sr,S),Y	7[1]	
STP	stop the processor	$DB	{implied}	3	. . . .
STX	store the X register	$8E	addr	4[7]	. . . .
		$86	dp	3[7]	
		$96	dp,Y	4[7]	
STY	store the Y register	$8C	addr	4[7]	. . . .
		$84	dp	3[7]	
		$94	dp,X	4[7]	

Mnemonic	*Instruction Operation*	*Opcode*	*Addressing Mode*	*Cycles*	*Effect on Flags* NV-BDIZC $e=1$ NVmxDIZC $e=0$
STZ	store zero	$9C	addr	4[1]	....
		$64	dp	3[1]	
		$74	dp,X	4[1]	
		$9E	addr,X	5[1]	
TAX	transfer A to X	$AA	{implied}	2	N.....Z.
TAY	transfer A to Y	$A8	{implied}	2	N.....Z.
TCD	transfer C to D	$5B	{implied}	2	N.....Z.
TCS	transfer C to stack pointer	$1B	{implied}	2	....
TDC	transfer D to C	$7B	{implied}	2	N.....Z.
TRB	test and reset bits	$1C	addr	6[2]	......Z.
		$14	dp	5[2]	
TSB	test and set bits	$0C	addr	6[2]	......Z.
		$04	dp	5[2]	
TSC	transfer stack pointer to C	$3B	{implied}	2	N.....Z.
TSX	transfer stack pointer to X	$BA	{implied}	2	N.....Z.
TXA	transfer X to A	$8A	{implied}	2	N.....Z.
TXS	transfer X to stack pointer	$9A	{implied}	2	....
TXY	transfer X to Y	$9B	{implied}	2	N.....Z.

					Effect on Flags
	Instruction		*Addressing*		NV-BDIZC *e*=1
Mnemonic	*Operation*	*Opcode*	*Mode*	*Cycles*	NVmxDIZC *e*=0
TYA	transfer Y to A	$98	{implied}	2	N.....Z.
TYX	transfer Y to X	$BB	{implied}	2	N.....Z.
WAI	wait for interrupt	$CB	{implied}	2	....
WDM	{reserved opcode}	$42	{implied}	2	....
XBA	exchange B and A	$EB	{implied}	3	N.....Z.
XCE[b]	exchange carry and emulation status bits	$FB	{implied}	2	C

NOTES ON CYCLE TIMES:
For all instructions involving direct page, add 1 cycle if the low-order byte of the direct page register is non-zero.

[1] Add 1 cycle if m=0.

[2] Add 2 cycles if m=0.

[3] Add 1 cycle if indexing across a bank boundary.

[4] Add 1 cycle if the branch is taken.

[5] Add 1 cycle if e=1 and the branch is taken across a page boundary.

[6] Add 1 cycle if e=0.

[7] Add 1 cycle if x=0.

[8] 7 cycles per byte moved.

OTHER NOTES:
In the "Effect on Flags" column, 1 means the flag is always set, 0 means the flag is always cleared, and a letter means that the flag changes depending on the result of the operation.

[a] In emulation mode, the B bit is set to 1 after a BRK instruction.

[b] XCE also affects the state of the *e* flag.

Listing 2-1: Useful Macros for Program Development

```
;
; PushWord Value
; PushWord #Value
;
; This macro pushes a word on the stack. If an
; address is specified ("Value"), this is done with
; LDA, PHA instructions. If an immediate number
; is specified ("#Value"), the number is put
; on the stack with a PEA instruction.
;
           MACRO
&LAB       PushWord &Value

           LCLC    &CHAR
&CHAR      AMID    &Value,1,1 ;Get 1st character
           AIF     "&CHAR"="#",.IMMEDIATE

&LAB       LDA     &Value
           PHA

           MEXIT

.IMMEDIATE

&CHAR      AMID    &Value,2,100

&LAB       PEA     &CHAR

           MEND


;
; PushLong Value
; PushLong #Value
;
; This macro pushes a long word on the stack. If an
; address is specified ("Value"), this is done with
; LDA, PHA, LDA, PHA instructions. If an immediate
; number is specified ("#Value"), the number is put
; on the stack with two PEA instructions.
;
           MACRO
&lab       PUSHLONG &Value

           LCLC    &CHAR
&CHAR      AMID    &Value,1,1 ;Get 1st character
           AIF     "&CHAR"="#",.IMMEDIATE
```

```
&lab        LDA      &Value+2
            PHA
            LDA      &Value
            PHA

            MEXIT

.IMMEDIATE
&CHAR       AMID     &Value,2,100

&lab        DC       I1'$F4'            ;PEA opcode
            DC       I2'(&CHAR)/-16'    ;Address high
            DC       I1'$F4'            ;PEA opcode
            DC       I2'&CHAR'          ;Address low

            MEND

;
; PushPtr Label
;
; Push a four byte pointer on the stack.
;
            MACRO
&lab        PushPtr &Label

&lab        DC       I1'$F4'            ;PEA opcode
            DC       I2'(&Label)|-16'   ;Address high
            DC       I1'$F4'            ;PEA opcode
            DC       I2'&Label'         ;Address low
            MEND

;
; PopLong DataAddr
;
; This macro pops a long word off the stack
; and stores it at DataAddr and DataAddr+2.
;
            MACRO
&lab        PopLong &DataAddr
&lab        PLA
            STA      &DataAddr
            PLA
            STA      &DataAddr+2
            MEND

;
; PopWord DataAddr
;
; This macro pops a word off the stack
; and stores it at DataAddr.
;
```

```
           MACRO
&lab       PopWord &DataAddr
&lab       PLA
           STA     &DataAddr
           MEND

;
; STR 'a string'
;
; Stores the specified character string in memory
; preceded by a length byte.
;
           MACRO
&LAB       STR     &String
&LAB       DC      I1'L:&String'
           DC      C"&String"
           MEND
```

CHAPTER 3

Using the GS Tools

Over the past several years, Apple has spent considerable time and energy developing and promoting what it considers to be the ideal method of communication between a computer user and a computer program. The chosen method, called a *user interface*, was first popularized on the Lisa (later renamed the Macintosh XL) and the Macintosh. It is based on a desktop metaphor which goes something like this:

- The screen is the desktop
- Pictorial icons on the screen represent objects on the desktop (such as file folders and clocks)
- Rectangular, overlappable windows represent papers on the desktop
- Pull-down menus at the top of the screen represent drawers in the desk

Such an interface requires a "hand" for quickly selecting papers and other objects on the desktop and for moving them around; the computer's hand is the mouse. To select an object with the mouse, move the mouse pointer over the object's icon and click (press and release) the mouse button. To drag an object to another part of the screen, move the mouse pointer over the object, press the mouse button, move the mouse while holding the button down, and then release the button when the object is where you want it to be.

A complete description of the desktop environment can be found in Apple Computer, Inc.'s *Human Interface Guidelines*. (See appendix 8.) Be sure to read this book before attempting to develop professional-quality software for the GS.

To promote the use of the desktop environment on the GS, Apple provides an extensive set of software tools that programmers can use to manage all aspects of the interface. Some of these tools are built into the GS ROM; others are loaded into RAM from disk when an application begins to run. The availability of these tools means most GS-specific programs will use them. As a result, users can concentrate on mastering the unique portions of a new program instead of its user interface.

As will be discussed below, there are also tools that perform standard operations more directly related to the operating system than to the desktop interface: mathematical calculations, interactions with I/O devices, memory management, and so on.

The GS tools are simply a series of 65816 subroutines called *functions.* For convenience, the functions are divided into several logical groups called *tool sets*; the functions in a particular tool set perform the same general type of operation: memory management, window handling, menu handling, and other tasks.

This chapter describes the tool sets that are available on the GS and shows you how to access them from within programs. The process of writing new tool sets from scratch and making them available to your applications is also covered.

TOOL SET SUMMARY

Table 3–1 shows what tool sets are available for the GS and gives the names of the corresponding APW macro definition files. It also indicates whether the tool sets are located in ROM or whether they must be loaded into RAM from disk by the application. (RAM-based tool sets must be stored in the SYSTEM/TOOLS/ subdirectory of the boot disk. See chapter 10.) Keep in mind that as bugs are eradicated, Apple will begin to move more of the RAM-based tools to ROM in order to free up disk space and improve program performance.

Later chapters will examine most of the common GS tool sets by describing their functions and showing how to use them in programs. This chapter merely summarizes the main features of each of the standard tool sets.

Tool Locator

The Tool Locator is responsible for the smooth, concurrent operation of all the GS tool sets. Most applications use it explicitly for only two reasons: to load RAM-based tool sets from disk and to install custom tool sets. Many tool sets use the Tool Locator implicitly, primarily to save a pointer to a general-purpose workspace in a Work Area Pointer Table (WAPT) maintained by the Tool Locator.

Memory Manager

An application uses the Memory Manager to allocate blocks of memory that have not previously been reserved by the operating system or another application. This means programmers no longer have to worry about memory conflicts. The other major function of the Memory Manager is to free up previously allocated blocks.

Table 3–1: The Standard Apple IIGS Tool Sets

Tool Set Number	*Tool Set Name*	*APW Macro File*
1	*Tool Locator	M16.LOCATOR
2	*Memory Manager	M16.MEMORY
3	*Miscellaneous Tool Set	M16.MISCTOOL
4	*QuickDraw II	M16.QUICKDRAW
5	*Desk Manager	M16.DESK
6	*Event Manager	M16.EVENT
7	*Scheduler	M16.SCHEDULER
8	*Sound Manager	M16.SOUND
9	*DeskTop Bus Tool Set	M16.ADB
10	*Floating-Point Numerics (SANE)	M16.SANE
11	*Integer Math Tool Set	M16.INTMATH
12	*Text Tool Set	M16.TEXTTOOL
13	*RAM Disk Tool Set	[internal use]
14	Window Manager	M16.WINDOW
15	Menu Manager	M16.MENU
16	Control Manager	M16.CONTROL
17	System Loader	M16.LOADER
18	QuickDraw Auxiliary Tool Set	M16.QDAUX
19	Print Manager	M16.PRINT
20	LineEdit	M16.LINEEDIT
21	Dialog Manager	M16.DIALOG
22	Scrap Manager	M16.SCRAP
23	Standard File Operations Tool Set	M16.STDFILE
24	Disk Utilities	[none]
25	Note Synthesizer	M16.NOTESYN

Table 3–1: Continued

Tool Set Number	*Tool Set Name*	*APW Macro File*
26	Note Sequencer	[none]
27	Font Manager	M16.FONT
28	List Manager	M16.LIST

NOTE: Tool sets marked by an asterisk (*) are in ROM.

Miscellaneous Tool Set

Applications do not often use the Miscellaneous Tool Set because it performs low-level tasks that are usually handled by the operating system, such as assigning ID tags to memory blocks, setting the date and time, and enabling interrupts.

QuickDraw II

QuickDraw II is the largest and most complex of the GS tool sets. Its main duties are to perform all drawing operations on the super high-resolution screen: activities such as plotting points, drawing lines, filling shapes, and displaying characters. In addition, it controls pen positioning, defines coordinate systems, and changes drawing parameters (such as color, pen size, and other characteristics).

Desk Manager

The Desk Manager allows you to install Classic Desk Accessories (CDAs) and New Desk Accessories (NDAs) in the system. A CDA is an accessory, such as the Control Panel, that you can call up by pressing Control-OpenApple-Esc from the keyboard. An NDA is an accessory that you can call up in a desktop environment by selecting its name from a pull-down menu.

Event Manager

Generally speaking, an event is the occurrence of a condition caused by an I/O device like the keyboard or the mouse. The most common events are the pressing of a key or the mouse button and the release of the mouse button. Two types of events are not tied to I/O devices at all—update events and activate events; these events occur when the appearance of a window needs to be changed. The Event Manager is responsible for keeping track of events and reporting them to the application when requested to do so.

Scheduler

The Scheduler provides a mechanism for ensuring that a busy, non-reentrant program module, such as ProDOS 16, will not be called by a program that gets control during a system interrupt until the module is no longer busy. The main use of this tool set is in scheduling the Control-OpenApple-Esc Classic Desk Accessory interrupt. As chapter 10 will discuss, you will also use it when installing certain types of interrupt-handling subroutines into ProDOS 16.

Sound Manager

The Sound Manager lets you control the behavior of the GS's Ensoniq DOC sound synthesizer. It also includes functions for accessing the 64K RAM area dedicated to the DOC.

DeskTop Bus Tool Set

The DeskTop Bus Tool Set lets you communicate with input devices connected to the Apple DeskTop Bus.

Floating-Point Numerics (SANE)

The Floating-point Numerics tool set implements the Standard Apple Numeric Environment. It is made up of a group of functions that applications can use to perform floating-point mathematical operations.

Integer Math Tool Set

The Integer Math Tool Set contains functions that permit the mathematical manipulation of integer numbers. This tool set also contains nine convenient number conversion utilities, which are described in appendix 4.

Text Tool Set

The Text Tool Set is important to programs that wish to use the text screen, rather than the super high-resolution graphics screen, for output. With this tool set, a program can print characters to the screen, much as traditional IIe software does. You can also use this tool set to redirect character output to a printer. The Text Tool Set is actually quite general: you can use it to send characters to any port or slot or to get input from any port or slot (or directly from the keyboard).

RAM Disk Tool Set

This tool set is for use by the operating system only; applications must not use it. It is called to control all operations related to the RAM Disk device you can create with the Control Panel.

Window Manager

The Window Manager is responsible for all activities related to windows: creating them, destroying them, dragging them around the screen, re-sizing them, and so on.

Menu Manager

The Menu Manager contains the functions an application needs to create and manage pull-down menus in the desktop environment.

Control Manager

A control is an object that can be selected to cause an immediate action or to set a parameter that will affect a future action. With the Control Manager you can associate various types of controls with a window. The main standard controls are push buttons, checkboxes, radio buttons, editable text, and scroll bars. The control manager also lets you define your own controls.

System Loader

The main function of the System Loader is to load a ProDOS 16 application (filetype S16) or any other type of ProDOS 16 load file from disk into memory so that it can be executed. It also takes care of two major preliminary steps: determining the size of the application and reserving (with the Memory Manager) a block of memory at which it can be loaded. (Some applications may require more than one block.) The System Loader is loaded into memory when you boot a ProDOS 16 disk.

QuickDraw Auxiliary Tool Set

The Quickdraw Auxiliary Tool Set contains additional functions for the QuickDraw II tool set.

Print Manager

The Print Manager is a set of functions that lets you send text and graphics to a printer instead of to the screen.

LineEdit

The Line Edit tool set lets you edit lines of text in a manner consistent with Apple's user-interface guidelines. It supports standard cut, copy, paste, delete, and range-selection operations.

Dialog Manager

The Dialog Manager controls the use of dialog and alert boxes. Programs use these boxes when they have important messages to display or when they want the user to enter information.

Scrap Manager

The Scrap Manager contains functions for transferring data to and from a data storage area called the clipboard. Using a clipboard makes it possible to easily transfer from one program to another, or even between different modules of the same program.

Standard File Operations Tool Set

The Standard File Operations Tool Set provides standard dialog boxes to be used when the applications need to know the name of a file to be opened or saved.

Note Synthesizer

The Note Synthesizer lets you program the Ensoniq DOC sound synthesizer to play musical notes with user-defined instruments.

Note Sequencer

The Note Sequencer has functions that let you play back a user-definable sequence of notes. The frequency and duration of the notes can be set by the application.

Font Manager

The Font Manager functions let you manage character fonts stored in the SYSTEM/FONTS/ directory on disk. With this tool set, an application can easily load and select fonts and choose font attributes.

List Manager

The List Manager lets you simplify the handling of lists of items. The items could represent filenames, strings, fonts, color patterns, or any other group of data elements that have the same row height on the graphics screen. The items in a list can be displayed in such a way that the user can move through the list using a vertical scroll bar.

USING THE TOOLS

Apple has developed a standard technique for calling any function of any tool set. To use this technique, the 65816 must be in full native mode with 16-bit X, Y, and A registers. If it is not, an error occurs.

Here is the procedure to follow:

1. Reserve a space on the stack for the results returned by the function (if any). You can do this with any instruction that pushes data on the stack, although PHA is just as convenient as any other.
2. Push on the stack all input parameters required by the function (if any). These parameters can be numeric constants (words or long words), pointers, or handles.
3. Load the X register with the ID number for the function:

   ```
   256 * (function number) + tool set number
   ```

   That is, put the tool set number in the low-order part of X and the function number in the high-order part.
4. Make a long subroutine call (with JSL) to the system tool dispatcher at $E10000. If calling a function for a user-defined tool set (see below), call the user tool dispatcher at $E10008 instead.

On return, any results are on the top of the stack and must be removed with one or more pull operations (PLA, PLX, or PLY).

If an error occurs, the tool dispatcher sets the carry flag and returns with an error code in the accumulator.

A specific example will make it easier to understand what is involved in calling a function. Suppose you want to use the TaskMaster function in the Window Manager tool set. This function has two input parameters and returns a word as a result. Here is how to call it:

```
PHA                  ;Create stack space for result (word)

PEA $FFFF            ;Push first input parameter (word)

PEA TaskRec|-16      ;Push second parameter (high)
PEA TaskRec          ; ... and (low).

LDX #$1D0E           ;Function (high) and tool set (low)
JSL $E10000          ;Call the system tool dispatcher

PLA                  ;Pop the result into the
STA TheResult        ;two bytes beginning at TheResult
```

It is important to understand precisely what is happening in this sequence of code. Because the function returns a two-byte result, the first step is to push space for it on the stack with a PHA instruction (in full native mode, this decrements the stack pointer by two bytes; if the function had returned a four-byte result, two PHA instructions would have been used to reserve space). Next, each input parameter is pushed on the stack in the order dictated by the function definition. In the example, the first parameter is a word-sized constant and is pushed with a PEA instruction. The second parameter is a long address, so it is pushed (high-order word first) with two successive PEA instructions.

The LDX instruction loads the X register with the tool set ($0E) and function ($1D) numbers so that the tool dispatcher, called with the JSL $E10000 instruction, knows the function to which you want to pass control.

Finally, the result is pulled from the stack with a PLA instruction. If the result was a two-word quantity, you would pull twice and the low-order word would come off the stack first.

Remember that when you use a function which returns a result, you must push space for the result on the stack before calling it, and you must remove the result when the function ends. Failure to follow these two rules will put the stack out of kilter and could eventually cause your program to crash.

Data Types Used by Functions

The input or output parameters of a function can be one of three sizes: byte, word (two bytes), or long word (four bytes). If a parameter represents a numeric quantity (as opposed to an address) its data type is said to be *integer* for a word-sized number, *Boolean* for a true/false word-sized number, and *long integer* for a long-word-sized number. A Boolean parameter is one that is true (non-zero) or false (zero).

Note that addresses passed to functions are always four bytes long even though the 65816 uses only three bytes (24 bits) to form a long address. The extra byte (always $00) makes sure that there is always an integral number of parameter words on the stack, which makes them easier to access with word-sized (16-bit) operations.

The data type of an address is either *pointer* or *handle*. By convention, symbolic names for such data types end in "Ptr" or "Hndl." A pointer is simply the address of a data structure somewhere in memory. A handle is the address, not of a data structure itself, but of a location that contains a pointer to the data structure. Pointers and handles are described in greater detail in the next chapter.

Tool Set Macros

Table 3–1 (see above) contains the names of the APW macro definition files for each GS tool set. You can extract the macro definitions your program needs with the MACGEN command prior to assembling the program. As explained in chapter 2, the purpose of the tool set macro files is to assign a standard symbolic name to the LDX/JSL calling sequence for each tool set function. By convention, all function names begin with an underscore character. The name _TaskMaster, for example, refers to function $1D in tool set $0D (the Window Manager).

APW also comes with a set of general-purpose macros in the M16.UTILITY file. Three of these macros (PushWord, PushLong, and PushPtr) are particularly useful for pushing parameters on the stack prior to making a tool set function call and are similar to the macros listed at the end of chapter 2.

To push a parameter, use PushWord (for words), PushLong (for long words), or PushPtr (for address pointers). PushWord and PushLong are able to push numbers stored at an address or immediate numbers:

```
PushWord   MyParm        ;Push the word at MyParm

PushWord   #10           ;Push a 10 (immediate)

PushLong   MySize        ;Push the long word at MySize

PushLong   #$7FFFFF      ;Push a $7FFFFF (immediate)

PushPtr    MyDataArea    ;Push the address of MyDataSize
```

Notice that the arguments for the two macros involving immediate numbers are preceded by a "#" sign.

At assembly time, these macros are expanded into the following code sequences:

```
LDA  MyParm
PHA                          ;PushWord MyParm

PEA  10                      ;PushWord #10

LDA  MySize+2
PHA
LDA  MySize
PHA                          ;PushLong MySize

PEA  $7FFFFF|-16
PEA  $7FFFFF                 ;PushLong #$7FFFFF

PEA  MyDataArea|-16
PEA  MyData                  ;PushPtr MyDataArea
```

Notice that the immediate forms of PushWord and PushLong push the constants on the stack with PEA instructions. This conveniently avoids destroying the contents of the A register (or any other register).

The PushPtr macro pushes the address specified in its argument onto the stack in the same way that the immediate form of PushLong pushes its numeric argument. In fact, the only difference between PushPtr and PushLong is that PushPtr's argument does not require a leading number sign.

The two macros listed in chapter 2 for removing results from the stack were PopWord (for words) and PopLong (for long words). The argument for each is the address at which the result is to be stored. For example, "PopWord TheResult" is equivalent to the following code sequence:

```
    PLA
    STA  TheResult            ;PopWord
```

and PopLong MyAddress is equivalent to:

```
    PLA
    STA MyAddress
    PLA
    STA MyAddress+2           ;PopLong
```

By using these five simple pull and pop macros and the tool set function macros, you can make your source code more understandable and easier to maintain. In addition, the chances of committing errors, such as pushing or pulling two-word quantities in the wrong order, are minimized.

THE TOOL LOCATOR

The first tool set you should know about is the Tool Locator because it manages all the other tool sets. The main functions in the Tool Locator are summarized in table R3–1 at the end of the chapter, and will be examined in more detail later in this chapter.

Only a few Tool Locator functions will ever be needed by most applications. The first, TLStartup, prepares the Tool Locator for operation and must be called before using the functions it supports or functions of any other tool set. TLShutDown "turns off" the Tool Locator and should be called just before your application ends, after the program calls the ShutDown functions of any other tools that have been started up. Neither TLStartup nor TLShutDown requires any input parameters, and neither return a result.

The LoadTools function loads RAM-based tool sets from disk into memory so that their functions can be called by an application. Its only parameter is a pointer to a tool set load table containing a list of the ID numbers and version numbers of the tool sets to be loaded. For this loading scheme to work properly, the tool sets must be stored in disk files having names of the form TOOLxxx, where xxx is the three-digit decimal number of the tool set. In addition, these files must be located in the SYSTEM/TOOLS/ subdirectory of the start-up disk.

Here is an example of how to use LoadTools to load twelve standard tool sets from disk:

```
              PushPtr LoadTable
              _LoadTools
              RTS
    LoadTable DC     I2'12'          ;Number of tool sets
              DC     I2'14,0'        ;Window Manager
```

```
                DC      I2'15,0'            ;Menu Manager
                DC      I2'16,0'            ;Control Manager
                DC      I2'18,0'            ;QuickDraw Auxiliary Tool Set
                DC      I2'19,0'            ;Print Manager
                DC      I2'20,0'            ;Line Edit
                DC      I2'21,0'            ;Dialog Manager
                DC      I2'22,0'            ;Scrap Manager
                DC      I2'23,0'            ;Standard File Operations
                DC      I2'25,0'            ;Note Synthesizer
                DC      I2'27,0'            ;Font Manager
                DC      I2'28,0'            ;List Manager
```

Notice how the tool set load table is constructed. The first entry is the number of tool set entries stored in the table. Following it are two words for each tool set. The first is the tool set number; the second is the minimum version number expected (a value of 0 means any version will do). LoadTools returns an error if it cannot locate the proper versions of all the tools listed. (As explained below, errors are recorded by setting the carry flag and putting an error code in the accumulator.)

If only one tool set has to be loaded, use LoadOneTool:

```
    PushWord TSNum          ;tool set number
    PushWord Version        ;version number
    _LoadOneTool
```

You may want to use LoadOneTool in situations where the application can function even if the specified tool set is not present. For example, you could use it to try to load the Print Manager; if the load fails (the carry flag is set), the application can disable all printing-related commands. If you load a large group of tool sets with LoadTools and get an error, you cannot tell which tool set is missing.

RAM-based tool sets loaded into memory with LoadTools or LoadOneTool remain active until you call TLShutDown at the end of the program or until you call UnloadOneTool. Whereas TLShutDown removes all RAM-based tools, UnloadOneTool removes only one:

```
    PushWord TSNum          ;tool set number
    _UnloadOneTool
```

Use UnloadOneTool when you are through using a given tool so that the memory it occupies will be freed up.

You cannot load RAM-based tools if the boot disk is not in a drive. If you try, you will get ProDOS error $45 (volume not found). To recover from this, you should use TLMountVolume to display a dialog box asking the user to insert the boot disk. The application can try again when the user clicks the OK button, or it can abort if the Cancel button is clicked.

An example of how to use TLMountVolume is given in the STANDARD.ASM program in listing 3–1. Because TLMountVolume requires QuickDraw and the Event Manager to be active, do not try to load tools until after you have started up QuickDraw and the Event Manager.

THE STRUCTURE OF A TOOL SET

Each GS tool set may contain up to 255 functions, numbered from 1 to 255. A function number of 0 is not allowed. By convention, the first eight functions must perform certain specific actions dictated by the operating system—other functions can do anything the designer of the tool set wants them to do.

Of the first eight functions, two are reserved for future expansion (#7 and #8). Here is what the other six standard functions do:

BootInit (#1). This function performs any preliminary initialization of the tool set which might be necessary. The system monitor firmware calls it when the system starts up or, if the tool is RAM-based, when the tool is first loaded from disk. Applications must not call this function.

Startup (#2). This function prepares the tool set for action. An application must call it before using any other function in the tool set. Many tool sets require input parameters for their Startup functions; a common parameter is a bank $00 address used by the tool set as the base of a private direct page.

ShutDown (#3). This function releases any memory space allocated by the tool set since start-up, including memory pointed to by the work area pointer table (the WAPT is described below). An application should call this function just before it ends.

Version (#4). This function returns the version number of the tool set (a word). It requires no input parameters, so the calling sequence is:

```
PHA                 ;space for result
_xxxVersion
PLA                 ;pop the version number result
```

The major version number is in the high byte and the minor version number is in the low byte of the result. If the tool set is in the prototype stage, the high-order bit of the high-order byte is set to 1.

Reset (#5). This function returns the tool set to a known, default state. It is called when a system reset occurs.

Status (#6). This function returns true (non-zero) if the tool set is active and false (zero) if it is not.

Of these six standard functions, only two of them, Startup and ShutDown, are really needed by most applications. In later chapters, when specific tool sets are discussed, these will be the only reserved functions mentioned. Keep in mind, however, that the BootInit, Version, Reset, and Status functions are always present—they just are not used often.

Almost every program you write for the GS will begin by loading RAM-based tool sets from disk and calling the Startup function for each tool set the program uses. When a program ends, it will call the ShutDown function for each tool set. To simplify these operations, you should develop a standard program subroutine like the one in the STANDARD.ASM program in listing 3–1 and include it in your program source file with the COPY directive.

STANDARD.ASM has two entry points, DoStartup and DoShutDown. Call DoStartup at the beginning of a program to load all tools and to start up tool sets in the proper order. (The order is important because some tool sets rely on the presence of others before they will work.) Call DoShutDown to shut down all the tool sets and exit the program. Depending on the program you are developing, you may want to modify STANDARD.ASM to add or eliminate the group of tool sets it uses. If you change the LoadTools tool table, be sure to set the leading count word ("11" in the example) to the proper value. You will also have to modify the tool set start-up and shut-down sequences.

As listed, STANDARD.ASM starts the application up in the 640-by-200 graphics display mode. To set up 320-by-200 graphics instead, change VidMode and XMaxClamp to $00 and 320, respectively.

Notice that STANDARD.ASM also assigns several global symbolic labels to numeric constants. This is convenient, because it forces you to use the same labels in all your programs, thus making them easier to read.

DEVELOPING YOUR OWN TOOL SET

Associated with each tool set is a table of pointers to each of its function handlers, in function number order:

Number of functions plus 1 (4 bytes)

Address of BootInit function minus 1 (4 bytes)

Address of Startup function minus 1 (4 bytes)

Address of ShutDown function minus 1 (4 bytes)

Address of Version function minus 1 (4 bytes)

Address of Reset function minus 1 (4 bytes)

Address of Status function minus 1 (4 bytes)

Address of Reserved function minus 1 (4 bytes)

Address of Reserved function minus 1 (4 bytes)

Address of Function #9 minus 1 (4 bytes)

.

.

.

Address of Function #n minus 1 (4 bytes)

This is called a function pointer table (FPT).

To install a new tool set, you must insert a pointer to its FPT in one of the tool pointer tables (TPT) maintained by the Tool Locator (there is a system TPT for standard system tools and a user TPT for user-defined tools). Do this with the Tool Locator's SetTSPtr function:

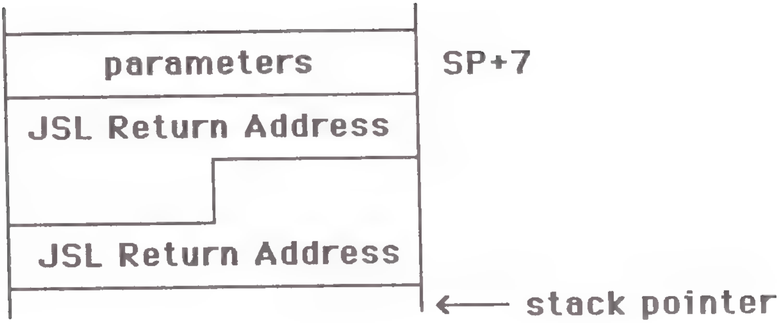
```
PushWord #$8000        ;8000=user, 0=system TPT
PushWord #243          ;This is the tool set number
PushPtr MyFPT          ;Pointer to new FPT
_SetTSPtr               ;Install pointer in TPT
```

The tool set number can be any number from 1 to 255 that is not already in use by another active tool set.

The first word pushed on the stack is the SystemOrUser word. It indicates whether you are dealing with system tool sets ($0000) or user-defined tool sets ($8000). Tool sets that are not just replacements for existing system tool sets should be added to the user TPT.

The function handlers pointed to by the FPT can be located anywhere in memory, but they cannot be allowed to move because the pointers to them would become invalid. The code for a function should be re-entrant, if possible; that is, the function should work even if it is interrupted and called by an interrupt handler or a Classic Desk Accessory. If it is not, the function must increment the Scheduler's busy flag when it gets control by performing a JSL $E10064 instruction in full native mode. If it does this, it must decrement the busy flag on exit with a JSL $E10068 instruction. Properly-designed interrupt handlers will not make tool set calls when the busy flag is non-zero.

When a function handler gets control, the 65816 is in full native mode, and the stack is configured as follows:



The last of the two 3-byte return addresses is caused by a JSL from the tool dispatcher to the function handler. The first is caused by the JSL to the tool dispatcher (at $E10000 or $E10008) itself.

This means that the parameters pushed on the stack by the application before it calls the tool dispatcher begin at SP+7 (7 bytes past the current stack pointer value). Note that the offset is 7, not 6, because SP points to the first byte past the last return address, not to the return address itself.

- **Warning!** These offsets remain valid only if the function handler does not push anything on the stack after gaining control. If it does, use offsets that take this into account.

To access parameters passed on the stack in this way, use the stack relative addressing mode, "sr,S". Suppose, for example, that the function has two input parameters, the first being a word and the second a long word, and the function returns a word result. To read the first parameter, use the instruction:

```
    LDA 11,S        ;Get 1st parameter
```

The offset of 11 is the sum of the basic offset, 7, and the space occupied by all parameters pushed after the first parameter (4 bytes for the single long word parameter in this example). The operands for accessing the second parameter would be 7,S (low-order word) and 9,S (high-order word).

The function handler returns a result by storing it directly into the space for the result on the stack. In this example, this space is at a location given by SP+13 (the address of the first parameter pushed plus the size of the first parameter).

Before it finishes, a function must remove all input parameters (but not the result) from the stack. It can do this by moving the two return addresses to positions x bytes higher in memory, where x is the total size of all the input parameters. The stack pointer must then be incremented by x bytes before the function ends. For instance, if there are 10 bytes of parameters, you would use the following code segment:

```
LDA     5,S             ;Move SP+1 through SP+6
STA     15,S            ; (the two return addresses)
LDA     3,S             ; up 10 bytes to SP+15
STA     13,S
LDA     1,S
STA     11,S

TSC                     ;Put SP in A register
CLC
ADC     #10             ;... add 10 to SP
TCS                     ;Save new SP
```

A function reports errors by setting the carry flag and storing an error code in the A register before ending with an RTL instruction. The error code actually occupies the lower 8 bits of the A register; the upper 8 bits hold the tool set number. The assignment of error conditions to error codes is up to the designer of the function.

If no error occurs, a function returns with the A register zeroed and the carry flag clear. A neat way of setting the carry flag properly, error or no error, is to exit with a CMP #1, RTL instruction sequence.

Note that a function need not preserve the contents of the X and Y registers. The direct page and data bank (and code bank) registers cannot change, however; they must be the same on exit as on entry. Of the status register bits, *m*, *x*, *e*, and *I* must be unchanged and the decimal mode flag must be zero.

Error Codes

Error codes from $0001 to $000F are reserved for use by the function dispatcher. Of these, only two are currently used:

$0001	The tool set does not exist
$0002	The tool set function does not exist

The function dispatcher may also return a $FFFF code; this means that the call to the dispatcher was not made in full native mode.

Error codes returned by functions themselves range from $xx01 to $xxFF, where xx represents the tool set number. One of these, $xxFF, has a special meaning, namely, “the function is not implemented.”

Work Areas

A tool set may require a private work area in which its functions can store configuration information and other data which must be saved between function calls. If the work area is not located in bank $00, function #1 of the tool set (BootInit) should reserve the space for it with the Memory Manager (see chapter 4).

Bank $00 areas are dealt with differently because they are prime real estate. Bank $00 is the only bank in which a direct page can be set up. By convention, if a tool set needs direct page space for a work area, the application must allocate it and pass its address to function #2 (Startup).

The tool set must place a pointer to its work area in a table called the Work Area Pointer Table (WAPT). (The pointer is zero if the tool set does not use the WAPT.) It can do this with the Tool Locator's SetWAP function. When a function handler takes control, the entry in the WAPT is always in the A (low-order word) and Y (high-order word) registers.

If the work area is in bank $00, the first thing the function handler should do is save the current contents of the direct page register on the stack and set the new direct page to the work area:

```
        PHD                 ;Save direct page register
        TCD                 ;"C" (16-bit A register) holds d.p.
```

Because this pushes an extra word on the stack, the address of the last parameter passed to the function is SP+9 and not SP+7.

Before returning to the tool dispatcher with an RTL, restore the previous direct page with a PLD instruction.

A USER-DEFINED TOOL SET

The program in listing 3–2 shows how to install a user-defined tool set called TimeTools. The program does this by passing a pointer to the tool set function pointer table to the SetTSPtr function. This table holds pointers to the eight standard functions every tool set must support and to one special function called DayOfWeek. You can use the DayOfWeek function to return a character string containing the name of the current day of the week.

The start-up function for the new tool set, TTStartup, expects a direct page address (a word) to be on the stack when it is called. It places this address in the WAPT using the SetWAP function. None of the TimeTools actually use this direct page area; the purpose of requiring it is simply to illustrate how to deal with work area pointers. The TTShutDown function clears the WAPT entry for the tool set to 0.

The TTStatus function returns a Boolean (word) result indicating whether the TimeTools are active. It returns true if the WAPT entry is non-zero (which means that TTStartup has been called).

The TimeTools tool set has one non-required function, DayOfWeek, which is assigned to function #9. It expects one input parameter: a pointer to a 10-byte buffer allocated by the application calling the function. DayOfWeek returns a character string in this buffer describing the day of the week.

DayOfWeek determines the day of the week by calling ReadTimeHex, a function in the Miscellaneous tool set that returns all time and date parameters in binary form (see chapter 5). It then takes the code for day of week (1 = Sunday, 2 = Monday, and so on), converts it to an ASCII-encoded string, and places it in the buffer, preceded by a length byte. It then removes the input parameter from the stack and returns.

Notice that GetDayOfWeek returns an error code of 1 if it receives a day of week code which is 0 or above 7. It does this by putting the error code in the accumulator and setting the carry flag by comparing the accumulator with 1.

Once the TimeTools tool set is installed, you can call its functions from anywhere inside the installation program. To call the DayOfWeek function, for example, use the following calling sequence:

```
PushPtr DOW_Buffer                 ;Pointer to buffer
LDX #$0901                         ;Function 9 / tool set 1
JSL $E10008                        ;(not E10000!)
RTS

DOW_Buffer DS   10                 ;Space for length + name
```

The program in listing 3–2 calls DayOfWeek in this way before displaying the day of the week on the 80-column text screen.

Notice that the JSL in this example is to location $E10008, the entry point to the user function dispatcher, not to location $E10000.

It is important to realize that you cannot use the TimeTools tool set from another application, because the memory it occupies is purged when the installation program in listing 3–2 ends. In any event, the user tool pointer table is cleared when the TLStartup or TLShutDown functions are called.

To allow a custom tool set to be used by any application, you must install it in the system tool pointer table. To do this, first choose a tool set number that is not already used by any other RAM- or ROM-based tool set. Then prepare a source file containing just the code defining the tool set. For the tool set in listing 3–2, for example, retain the ToolSet code segment only, change TS_NUM to the selected tool set number and change UserOrSys to $0000 to mark this as a system tool set.

Next, assemble and link the source file and then change the file type code of the EXE file created to $BA using the APW FILETYPE command. (ProDOS 16 recognizes only $BA files as system tool sets.) Finally, transfer the $BA file to the SYSTEM/TOOLS/ directory on the boot disk. After loading the tool set with LoadTools or LoadOneTool, applications can use it.

Table R3–1: The Major Functions of the Tool Locator Tool Set ($01)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
GetFuncPtr	$0B	result (L)	ptr to function handler
		SystemOrUser (W)	0 = system/$8000 = user
		FuncTSNum (W)	function (high)/tool set (low)
GetTSPtr	$09	result (L)	ptr to function pointer table
		SystemOrUser (W)	0 = system/$8000 = user
		TSNumber (W)	tool set number
GetWAP	$0C	result (L)	ptr to tool set work area
		SystemOrUser (W)	0 = system/$8000 = user
		TSNumber (W)	tool set number
LoanOneTool	$0F	TSNumber (W)	tool set number to load
		MinVersion (W)	minimum version required
LoadTools	$0E	ToolTable (L)	ptr to tool set load table
SetTSPtr	$0A	SystemOrUser	0 = system/$8000 = user
		TSNumber (W)	tool set number
		FPTptr (L)	ptr to function pointer table
SetWAP	$0D	SystemOrUser (W)	0 = system/$8000 = user
		TSNumber (W)	tool set number
		WAPptr (L)	ptr to work area
TLMountVolume	$11	result (W)	1 = OK, 2 = Cancel selected
		WhereX (W)	upper left-hand X coordinate
		WhereY (W)	upper left-hand Y coordinate
		Line1Ptr (L)	ptr to string at top of box
		Line2Ptr (L)	ptr to string below line 2

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
		Buttom1Ptr (L)	ptr to text for OK button
		Button2Ptr (L)	ptr to text for Cancel button
TLShutDown	$03	[no parameters]	
TLStartup	$02	[no parameters]	
UnloadOneTool	$10	TSNumber (W)	tool set number to unload

Table R3–2: Tool Locator Error Codes

Error Code	*Description of Error Condition*
$0110	The specified minimum tool set version was not found.

NOTE: LoadTools and LoadOneTool can also return ProDOS error codes.

Listing 3–1: The STANDARD.ASM Program with Standard Tool Start-up and Shut-down Sequences

```
;
; DoStartup and DoShutdown are standard opening
; and closing sequences for most programs.
;
; Set VidMode and XMaxClamp as appropriate for the
; video mode you're using (320x200 or 640x200).
;
; All the tools referred to in ToolTable must be in
; the SYSTEM/TOOLS/ directory. If you're missing some,
; remove the entry from the table, change the initial
; count byte, and remove the Startup and ShutDown calls.

DoStartup  START

* Direct page equates

DeRefLoc    GEQU    $0                ;LONG (used for dereferencing handle)
DPAddress   GEQU    DeRefLoc+4        ;INTEGER (address of direct page)

* General equates:

False       GEQU    $0000             ;Boolean false
True        GEQU    $8000             ;Boolean true

VidMode     GEQU    $80               ;$80 = 640x200, $00 = 320x200
XMaxClamp   GEQU    640               ;Video width (640 or 320)

* Point offsets:

v           GEQU    0                 ;Vertical position
h           GEQU    2                 ;Horizontal position

* Rectangle offsets:

top         GEQU    0
left        GEQU    2
bottom      GEQU    4
right       GEQU    6

* GetNextEvent and TaskMaster result codes:

btnDownEvt  GEQU    1                 ;mouse-down event
keyDownEvt  GEQU    3                 ;key-down event
autoKeyEvt  GEQU    5                 ;auto-key event
updateEvt   GEQU    6                 ;update event
activateEvt GEQU    8                 ;activate event
wInMenuBar  GEQU    17                ;in menu bar
wInContent  GEQU    19                ;in content region of window
```

```
wInGoAway  GEQU    22               ;in close box
wInSpecial GEQU    25               ;in special menu item

; Dialog Item Type codes:

ButtonItem GEQU    10
CheckItem  GEQU    11
RadioItem  GEQU    12
ScrollBarItem GEQU 13
UserCtlItem GEQU   14
StatText   GEQU    15
LongStatText GEQU  16
EditLine   GEQU    17
IconItem   GEQU    18
PicItem    GEQU    19
UserItem   GEQU    20
UserCtlItem2 GEQU  21

           Using   StartData

           PHK
           PLB                      ;Data bank = program bank

           _TLStartup               ;Tool Locator
           _MTStartup               ;Miscellaneous Tools

           PHA                      ;space for result
           _MMStartup               ;Memory Manager
           PopWord MyID

; Get direct page memory for the tool sets:

           PHA                      ;Space for handle
           PHA
           PushLong #$D00           ;Thirteen pages
           PushWord MyID            ;ID tag to use
           PushWord #$C005          ;Locked, fixed, aligned, fixed bank
           PushLong #0              ;Bank $00
           _NewHandle
           PopLong  DeRefLoc

           LDA     [DeRefLoc]       ;Dereference the handle
           STA     DPAddress        ; and save the pointer (low)

           PushWord DPAddress       ;DP to use (3 pages)
           PushWord #VidMode        ;Graphics mode
           PushWord #160            ;max width
           PushWord MyID            ;ID tag to use
           _QDStartup               ;QuickDraw II

           _InitCursor              ;Display arrow cursor
```

```
          LDA      DPAddress
          CLC
          ADC      #$300              ;one page
          PHA
          PushWord #0                 ;queue size (default)
          PushWord #0                 ;x min for clamp
          PushWord #XMaxClamp         ;x max for clamp
          PushWord #0                 ;y min for clamp
          PushWord #200               ;y max for clamp
          PushWord MyID               ;ID tag to use
          _EMStartUp                  ;Event Manager

* Now load the RAM-based tools:

GetTools  PushPtr ToolTable
          _LoadTools
          BCC      StartMore

          CMP      #$45               ;Error was Volume Not Found?
          BNE      Abort              ;No, so branch

          JSR      AskForDisk         ;Ask for boot volume
          CMP      #1                 ;OK?
          BEQ      GetTools           ;Yes, so branch

Abort     PLA                         ;(remove return address)
          JMP      DoShut1

;Start up the rest of the tool sets:

StartMore  ANOP

          _QDAuxStartup               ;QuickDraw Auxiliary Tools

          PushWord MyID
          LDA      DPAddress
          CLC
          ADC      #$400              ;one page
          PHA
          _CtlStartup                 ;Control Manager

          PushWord MyID
          _WindStartup                ;Window Manager

          PushLong #0
          _RefreshDesktop             ;Draw the screen
```

```
            PushWord MyID
            LDA      DPAddress
            CLC
            ADC      #$500              ;one page
            PHA
            _MenuStartup                ;Menu Manager


            PushWord MyID
            LDA      DPAddress
            CLC
            ADC      #$600              ;one page
            PHA
            _LEStartup                  ;LineEdit


            PushWord MyID
            _DialogStartup              ;Dialog Manager


            PushWord MyID
            LDA      DPAddress
            CLC
            ADC      #$700              ;one page
            PHA
            _SFStartup                  ;Standard File Operations


            LDA      DPAddress
            CLC
            ADC      #$800              ;one page
            PHA
            _SoundStartup               ;Sound Manager

; Insert this to start up the Print Manager:


;           PushWord MyID
;           LDA      DPAddress
;           CLC
;           ADC      #$900              ;two pages
;           PHA
;           _PMStartup                  ;Print Manager


            PushWord MyID
            LDA      DPAddress
            CLC
            ADC      #$B00              ;one page
            PHA
            _FMStartup                  ;Font Manager
```

```
           LDA      DPAddress
           CLC
           ADC      #$C00              ;one page
           PHA
           _SANEStartup                ;Numerics (SANE)

           _ScrapStartup               ;Scrap Manager
           _IMStartup                  ;Integer Math
           _DeskStartup                ;Desk Manager

           RTS

AskForDisk ANOP

           _GET_BOOT_VOL GBVParms      ;Get name of boot volume

           PHA                         ;space for result
           PushWord #150
           PushWord #50
           PushPtr Prompt1
           PushPtr VolName
           PushPtr OK_Msg
           PushPtr Cancel_Msg
           _TLMountVolume
           PLA
           RTS

GBVParms   DC       I4'VolName'

Prompt1    STR      'Insert the volume:'
VolName    DS       18                 ;Space for volume name

OK_Msg     STR      'OK'
Cancel_Msg STR      'Abort'


DoShutDown ENTRY

           _DeskShutDown
           _IMShutDown
           _ScrapShutDown
           _SANEShutDown
           _FMShutDown
;          _PMShutDown                 ;Call this if using Print Manager
           _SoundShutDown
           _SFShutDown
           _DialogShutDown
           _LEShutDown
           _MenuShutDown
```

```
          _WindShutDown
          _CtlShutDown
          _QDAuxShutDown

DoShut1   ENTRY                     ;Enter here if LoadTools fails

          _EMShutDown
          _QDShutDown

          PushWord MyID
          _MMShutDown

          _MTShutDown
          _TLShutDown

          _Quit   QuitParms

          BRK     $F0               ;(Shouldn't get this far)

          END

StartData DATA

MyID      DS      2

QuitParms DC      I4'0'             ;Return to caller
          DC      I2'0'             ;No special flags

; Table of RAM-based tools to load. Only load the tools you
; have in the SYSTEM/TOOLS/ directory.

ToolTable DC      I'11'             ;Number of tools to load
          DC      I'14,$0000'       ;Window Manager
          DC      I'15,$0100'       ;Menu Manager
          DC      I'16,$0000'       ;Control Manager
;         DC      I'17,$0000'       ;(System Loader always loaded)
          DC      I'18,$0000'       ;QuickDraw Aux Tools
;         DC      I'19,$0000'       ;Print Manager
          DC      I'20,$0000'       ;Line Edit
          DC      I'21,$0000'       ;Dialog Manager
          DC      I'22,$0000'       ;Scrap Manager
          DC      I'23,$0000'       ;Standard File Operations
;         DC      I'24,$0000'       ;Disk Utilities
          DC      I'25,$0000'       ;Note Synthesizer
;         DC      I'26,$0000'       ;Note Sequencer
          DC      I'27,$0000'       ;Font Manager
          DC      I'28,$0000'       ;List Manager

          END
```

Listing 3–2: A User-defined Tool Set

```
***********************************************
* This program shows what a custom tool set   *
* looks like. The first portion installs      *
* the tool set into the user tool set table.  *
***********************************************
           KEEP    TOOLSET
           MCOPY   TOOLSET.MAC

TS_NUM     GEQU    $01                 ;Tool set number
UserOrSys  GEQU    $8000               ;$8000 = user tool set

Installer  START

           _TLStartup
           _TextStartup

; This code installs the user tool set:

           PushWord #UserOrSys         ;Tool set type
           PushWord #TS_NUM            ;Tool set number
           PushPtr  ToolSet            ;Pointer to FPT
           _SetTSPtr                   ;Install the tool set

; Now let's test the tool set by showing the day of the week:

           PushPtr TheBuffer
           LDX #TS_NUM+9*256           ;Call DayOfWeek function
           JSL $E10008                 ;$E10008 = user dispatcher

           PushPtr TheBuffer
           _WriteString                ;Display on text screen

           _TextShutDown
           _TLShutDown                 ;(this removes user tool sets)

           _QUIT   QuitParms

           BRK     $F0

QuitParms  DC      I4'0'
           DC      I2'0'

TheBuffer  DS  10                      ;Day name returned here

           END

* The tool set definition begins here:

ToolSet    START
```

```
* This is the function pointer table. It contains the addresses,
* minus 1, of each function subroutine. It begins with a long
* word containing the number of functions, plus 1.

            DC      I4'(TBL_END-ToolSet)/4'   ;Number of functions (+1)
            DC      I4'TTBootInit-1'
            DC      I4'TTStartup-1'
            DC      I4'TTShutDown-1'
            DC      I4'TTVersion-1'
            DC      I4'TTReset-1'
            DC      I4'TTStatus-1'
            DC      I4'Reserved-1'
            DC      I4'Reserved-1'
            DC      I4'DayOfWeek-1'

TBL_END     ANOP

TTBootInit  LDA     #0                  ;Null error code
            CLC                         ;No error
RTL

;TTStartup has one input parameter: a word representing a starting
; address in bank $00 of a one-page work area. On entry to TTStartup,
; this word is buried $07 bytes into the stack, just above the two
; 3-byte return addresses. On exit, this function removes the input
; parameter from the stack by moving the two return addresses and the
; stack pointer up by two bytes.

TTStartup   ANOP

            LDA     $07,S               ;Get direct page address

            PushWord #UserOrSys         ;Tool set type
            PushWord #TS_NUM            ;Tool set number
            PushWord #0                 ;High word of address (zero)
            PHA                         ;Low word of address
            _SetWAP

; Remove the input parameter from the stack:

            LDA     5,S                 ;Move the two long JSL
            STA     7,S                 ; return address up by
            LDA     3,S                 ; two bytes
            STA     5,S
            LDA     1,S
            STA     3,S

            TSC                         ;Increment the SP by
            CLC                         ; two bytes.
            ADC     #2                  ;(Could just do PLA instead)
            TCS
```

```
           LDA      #0                 ;Exit with no error
           CLC
           RTL

;To shutdown this tool set, just zero the entry in the
; work area pointer table. It is up to the application to
; dispose of the area it points to (using DisposeHandle).

TTShutDown ANOP

           PushWord #UserOrSys         ;Tool set type
           PushWord #TS_NUM            ;Tool set number
           PushLong #0                 ;Address is zero
           _SetWAP

           LDA      #0
           CLC
           RTL

TTVersion returns a version word in a space on the
; stack allocated by the caller. The high-order byte
; contains the main version number and the low-order
; byte contains the secondary version number. The
; stack result space begins at an offset of $07 from
; SP because there are two 3-byte return address on
; the top of the stack on entry to TTVersion.

TTVersion  ANOP

           LDA      #$0201             ;Version 2.1
           STA      $07,S              ;Put result in stack space

           LDA      #0
           CLC
           RTL

TTReset    LDA      #0
           CLC
           RTL

TTStatus returns a Boolean (word) in the stack space
; reserved by the caller. The result is true if TTStartup
; has been called; false otherwise. TTStatus knows if it
; has been called because the WAPT entry will be nonzero.

TTStatus   ANOP

           CMP      #0                 ;Is WAP (low) zero?
           BEQ      SetStatus          ;Yes, so store false

           LDA      #$FFFF             ;True code
```

```
SetStatus  STA      $07,S                 ;Put result in stack space

           LDA      #0
           CLC
           RTL

Reserved   LDA      #TS_NUM*256+$FF       ;Error $FF means "not implemented
           SEC
           RTL

* The DayOfWeek function expects one parameter on the stack: a
* pointer to a data area where the function is to return the
* day-of-week string. Note that after the initial PHD and
* aligning the new direct page with the stack, the pointer
* is at [$09].
DayOfWeek  ANOP

OldDP      EQU      $01
RTL1       EQU      OldDP+2
RTL2       EQU      RTL1+3
ThePtr     EQU      RTL2+3

           PHD                            ;Save current direct page

           TSC                            ;Align d.p. with stack
           TCD

           PHA                            ;Space for 8 bytes of result
           PHA
           PHA
           PHA
           _ReadTimeHex
           PLA                            ;Pop minute/second
           PLA                            ;Pop year/hour
           PLA                            ;Pop month/day
           PLA                            ;Pop day of week (high byte)

           XBA                            ;Put day of week in low byte
           AND      #$0F                  ;Strip unused bits
           DEC      A                     ;Convert 1..7 to 0..6 (1=Sunday)

           CMP      #7                    ;Number in range (0..6)?
           BCS      DOW_Error             ;No, so branch

; Look for the Nth entry in the table:

           TAY
           LDX      #0
FindEntry  CPY      #0                    ;At correct name?
           BEQ      SaveResult            ;Yes, so branch
```

```
FE1         LDA     >DayTable,X         ;(Force long because B<>K)
            BEQ     FE2                 ;Branch if at end of name
            INX                         ;Move to next character
            BRA     FE1

FE2         INX                         ;Move to start of next name
            INX
            DEY                         ;Decrement day-of-week counter
            BRA     FindEntry

; Transfer the name to the buffer area:

SaveResult  SEP     #$20                ;Use 8-bit accumulator
            LONGA   OFF

            LDY     #1
SR1         LDA     >DayTable,X
            BEQ     SR2                 ;Branch if at end of name
            STA     [ThePtr],Y
            INX
            INY
            BRA     SR1

SR2         DEY
            TYA
            STA     [ThePtr]            ;Store length of string

            REP     #$20                ;Back to 16-bit accumulator
            LONGA   ON

            LDA     #0                  ;No error
            BRA     DOW_Exit

DOW_Error   LDA     #TS_NUM*256+$01     ;Error code #1

DOW_Exit    TAX                         ;Save error code in X

            PLD                         ;Restore direct page

; Remove the long input parameter from the stack:

            LDA     5,S                 ;Move the two long JSL
            STA     9,S                 ; return address up by
            LDA     3,S                 ; four bytes
            STA     7,S
            LDA     1,S
            STA     5,S

            TSC                         ;Increment the SP by
            CLC                         ; four bytes.
            ADC     #4
            TCS
```

```
          TXA                       ;Get error code
          CMP     #1                ;Set carry if error
          RTL

          MSB     OFF

DayTable  DC      C'Sunday',I2'0'
          DC      C'Monday',I2'0'
          DC      C'Tuesday',I2'0'
          DC      C'Wednesday',I2'0'
          DC      C'Thursday',I2'0'
          DC      C'Friday',I2'0'
          DC      C'Saturday',I2'0'

          END
```

CHAPTER 4

Memory Management

As discussed in chapter 2, the 65816 microprocessor that controls the GS can access directly an enormous 16-megabyte memory space (arranged as 256 memory banks of 64K each); the 24-bit memory addresses run from $000000 to $FFFFFF. This space can be made up of any combination of ROM, RAM, and memory-mapped I/O (soft switch) locations, but it need not be fully populated. In a computer system, the ROM holds a bare-bones operating-system program (sometimes called a *system monitor*) which loads a more elaborate operating system from disk when the system is turned on. It also usually contains a collection of subroutines that an application program can call to perform standard tasks, such as clearing the screen, reading the keyboard, and printing a character. RAM, of course, is where an application stored on disk is loaded and where data of any sort can be stored by an application or the operating system.

This chapter investigates the special uses the GS makes of the memory in the 65816's address space. Some clues concerning the operation of the GS will be uncovered, and you will discover which areas of RAM are available for use by your own applications.

When writing an application for the GS (or any computer for that matter), it is important to avoid using memory areas already in use by the operating system or by any other application. If you overwrite busy memory areas, the system is sure to behave unpredictably.

It is easy to write a program for the IIe that does not conflict with the original ProDOS 8 operating system. ProDOS 8 occupies specific memory areas and programmers are responsible enough not to use those areas for applications. Trying to live with another application in memory, such as a desk accessory utility, can be fatal, however. Why? Because no software protocol for informing an application of the memory other applications are occupying has been implemented. A co-resident desk accessory utility, such as the Pinpoint Desk Accessories, is useful only because it occupies an area few applications use (the auxiliary bank-switched RAM area).

A program can avoid memory conflicts in a ProDOS 16 GS environment simply by using a tool set called the Memory Manager. When the program needs a block of memory to work with, it simply issues a request to the Memory Manager. The Memory Manager processes the request by locating a free area of memory of the proper size, marking it as in use, and returning its handle (the address of a pointer to it). As long as every program uses the Memory Manager, no program will tread on the toes of any other program.

The use of the Memory Manager is discussed at the end of this chapter.

GS MEMORY MAP

As shown in figure 4–1, the GS can address up to 256 banks of memory. The banks are numbered from $00 to $FF, and each one is 64K in size. In its minimal configuration (no card in the memory expansion slot), the GS has 256K of RAM and 128K of ROM, so it uses only six of these banks: banks $00, $01, $E0, $E1, $FE, and $FF. The first four of these banks hold the RAM, and banks $FE and $FF hold the ROM. The rest of the banks are unoccupied or unavailable.

The RAM in banks $E0 and $E1 deserves special mention because it has one restrictive characteristic that the RAM in other banks does not: read and write operations affecting these banks always take place at the normal (1 MHz) clock speed, even if the system speed is set to fast (2.8 MHz).

If you are operating in fast mode and you start using either of these banks, the GS hardware slows the system down and automatically returns to fast mode when you are done. The reason for the slowdown is that banks $E0 and $E1 contain video display buffers and other memory-mapped I/O locations that, for reasons of compatibility with existing IIe software and hardware, must be accessed at the same clock speed used by the IIe (1 MHz).

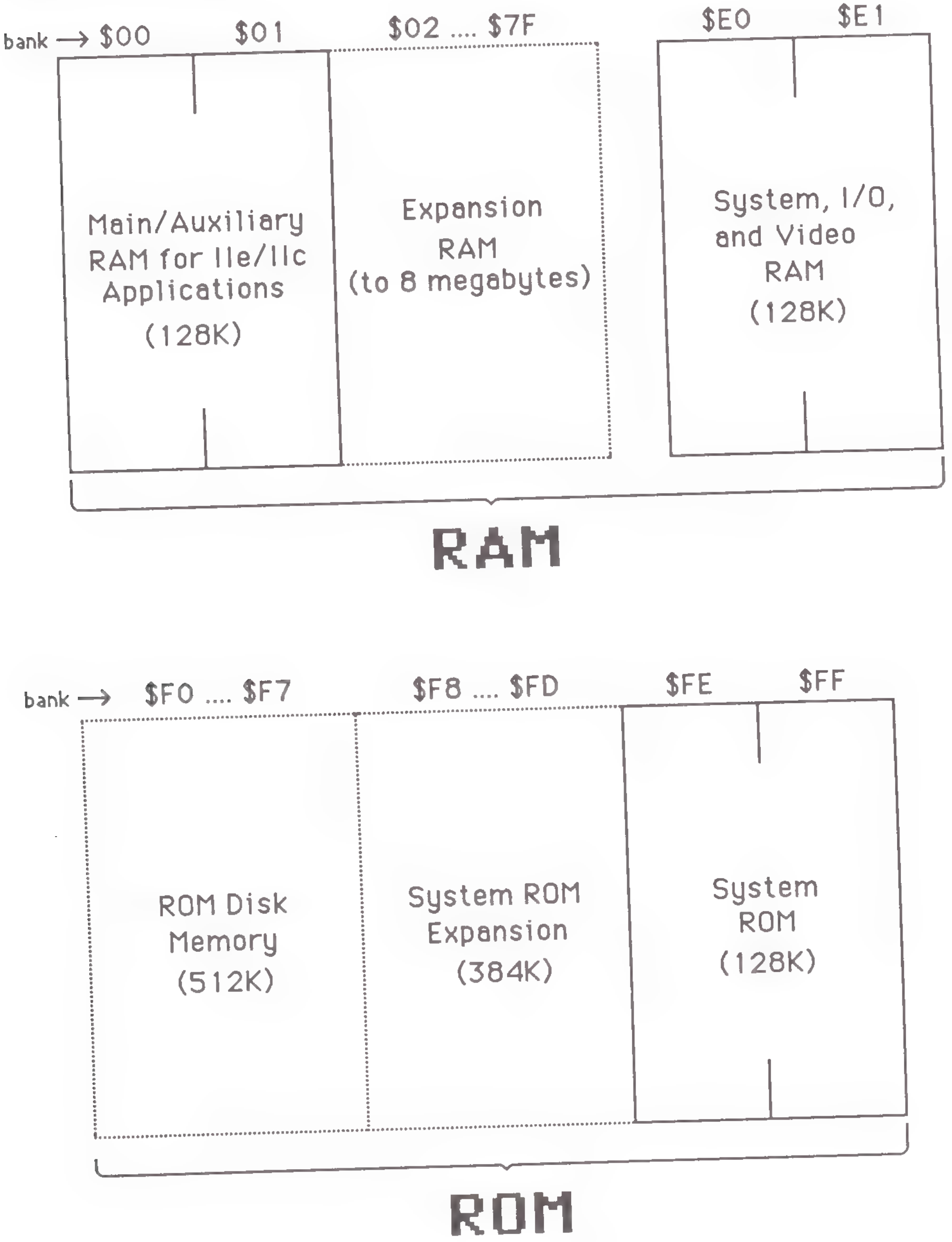
You can add more RAM to the GS by inserting an appropriate card in the memory expansion slot (see appendix 6). Additional RAM occupies consecutive banks beginning at bank $02—if you add one megabyte of memory (sixteen 64K banks), for example, it will occupy banks $02 through $11. The upper limit for RAM expansion through the memory expansion slot is 8 megabytes.

A memory expansion card may also contain ROM. The ROM may occupy any of the banks from $F0 to $FD. The data in banks $F0 to $F7 (a 512K space) is expected to be organized just as it would be on a disk; that is, these ROM banks are configured as a ROM Disk. The ROM in banks $F8 to $FD is designed to hold extensions to the firmware ROM.

SPECIAL RAM AREAS

Several areas in the four core RAM banks ($00, $01, $E0, and $E1) are used for special purposes, by the 65816, the operating system, the firmware, or I/O devices. Programs must respect this usage and not attempt to use these areas for other purposes, except where the context permits.

Figure 4–1. The Apple IIgs Memory Map (Each RAM bank is 64K in size.)


bank → $00
$01
$02 .... $7F
$E0
$E1
Main/Auxiliary RAM for IIe/IIc Applications (128K)
Expansion RAM (to 8 megabytes)
System, I/O, and Video RAM (128K)
RAM
bank → $F0 .... $F7
$F8 .... $FD
$FE
$FF
ROM Disk Memory (512K)
System ROM Expansion (384K)
System ROM (128K)
ROM

Banks $E0 and $E1

The two RAM banks in the high end of memory, banks $E0 and $E1, are often referred to as the *firmware RAM* because the ROM drivers for peripheral devices, such as the serial ports, the mouse, and the disk, use these banks for data storage. The System Loader (the portion of the operating system that loads ProDOS 16 applications), AppleTalk, and tool sets also make extensive use of the firmware RAM. As mentioned earlier, reads and writes to banks $E0/$E1 always take place at 1 MHz, even if the system clock speed is set to 2.8 MHz.

The arrangement and use of memory in firmware RAM on the GS is, in many respects, the same as on the IIe. Bank $E0 looks much like main memory on the IIe and bank $E1 looks much like auxiliary memory. The similarities are reviewed below.

Language Card. The space from $D000 to $FFFF (12K in size) in either of the two banks of firmware RAM is called a *language card* (or bank-switched RAM) and can be associated with either 12K of ROM or 16K of RAM. A program can switch between ROM and RAM on the fly by manipulating a set of software-controllable switches (called *soft switches*).

The ROM for the language card is actually physically located in bank $FF (at the same relative position within the bank), but this area also maps to banks $E0 and $E1 when the GS is turned on. The ROM contains Applesoft, the system monitor, I/O drivers, tool sets, and more.

The RAM for the language card actually takes up 4K more space than what is available! The extra 4K is treated as a second $D000-$DFFF memory segment and soft switches are available to select which of these two banks is to be active. On the GS, the language card RAM in banks $E0/$E1 is reserved for use by the ProDOS System Loader and by AppleTalk.

I/O Space. The I/O space runs from $C000 to $CFFF. The first part of it, from $C000 to $C0FF, is actually made up of memory-mapped I/O locations—that is, locations that can be read from or written to so that you can communicate with I/O devices. Each slot (or built-in port) has the exclusive use of sixteen unique I/O locations: $C090-$C09F for slot 1, $C0A0-$C0AF for slot 2, and so on up to $C0F0-$C0FF for slot 7. The I/O locations from $C000 to $C07F control internal I/O devices (such as the keyboard and the video) and modes of operation (including memory selection, system speed, and graphics mode selection).

Peripheral ROM. The locations from $Cn00-$CnFF (n=1 to 7) are reserved for ROM found on a peripheral card plugged into slot *n*. (Internal ports have ROM associated with them, too. It is stored in the lower part of bank $FF but is also mapped to the $Cnxx pages in banks $E0 and $E1.) A typical ROM contains a program that provides the subroutines a program needs to communicate easily with a peripheral device. Each peripheral may also contain a 2K ROM area that occupies $C800-$CFFF; the peripheral can select this ROM area to the exclusion of all others.

While a program is running in the peripheral ROM area, the GS always operates at normal speed (1 MHz). This is done to permit timing-sensitive peripherals designed for the IIe to work properly on the GS. For example, an internal modem card like the Hayes Micromodem will not work at a clock speed other than 1 MHz because the program in its ROM that dials the telephone uses precise timing loops that are based on a 1 MHz clock speed.

Video Buffers. The video buffers for the standard IIe video display modes—40- and 80-column text, single- and double-width low-resolution and high-resolution graphics—are located in the same relative positions on the GS as on the IIe:

> $400-$7FF are used for the page 1 40-column text display and low-resolution graphics modes
>
> $800-$BFF are used for the page 2 40-column text display and low-resolution graphics modes
>
> $2000-$3FFF are used for the page 1 high-resolution graphics mode
>
> $4000-$5FFF are used for the page 2 high-resolution graphics mode

In each case, the standard text and graphics modes (40-column text and single-width graphics) use areas in bank $E0 only. For 80-column text and double-width graphics modes, the same areas in bank $E1 are also used.

Other Special Areas. Another display mode, called super high-resolution graphics, has a video buffer located from $2000 to $9FFF in bank $E1 only. This is the display buffer used by the QuickDraw II graphic drawing tool set that is stored in the GS ROM. It does not exist on the IIe.

The other areas in banks $E0 and $E1 that have special significance are the spaces from $0000 to $1FFF in both banks (which include the video buffers from $400 to $BFF that were discussed). These spaces are reserved by the operating system for the storage of variables, vectors, jump tables, and other data used by the GS tools and built-in I/O ports.

Free Areas. The areas in banks $E0/$E1 not used for special purposes are:

- $6000-$BFFF in bank $E0
- $A000-$BFFF in bank $E1

These areas are free for allocation by the Memory Manager. Keep in mind that, for all practical purposes, the video buffers used by page 1 and page 2 of standard high-resolution graphics ($2000-$5FFF in bank $E0) are also available if a program uses only the super high-resolution graphics mode.

Banks $00 and $01

Bank $00 is of particular importance, because this is where the 65816 locates its stack and direct page. In emulation mode, the stack occupies page $01 and the direct page occupies page $00. In native mode, however, the operating system can position the stack and direct page anywhere in bank $00.

One of the primary design constraints for the GS was that it work properly with existing IIe software (and hardware). To allow this, it was necessary to permit the remapping of banks $00 and $01 so that these areas could be used as the main and auxiliary memory areas are on the IIe, complete with an I/O space and a language card area. At the same time, I/O and video operations had to take place at the same clock speed as the IIe, namely 1 MHz.

One way to accomplish this might have been to force banks $00 and $01 to operate at 1 MHz all the time and to keep the standard video buffers and memory-mapped I/O locations in those banks. This would mean, however, that one especially common 65816 operation—pushing data on the stack (a bank $00 operation)—would always cause the system to slow down, severely restricting the advantage of operating the GS in 65816 native mode at 2.8 MHz.

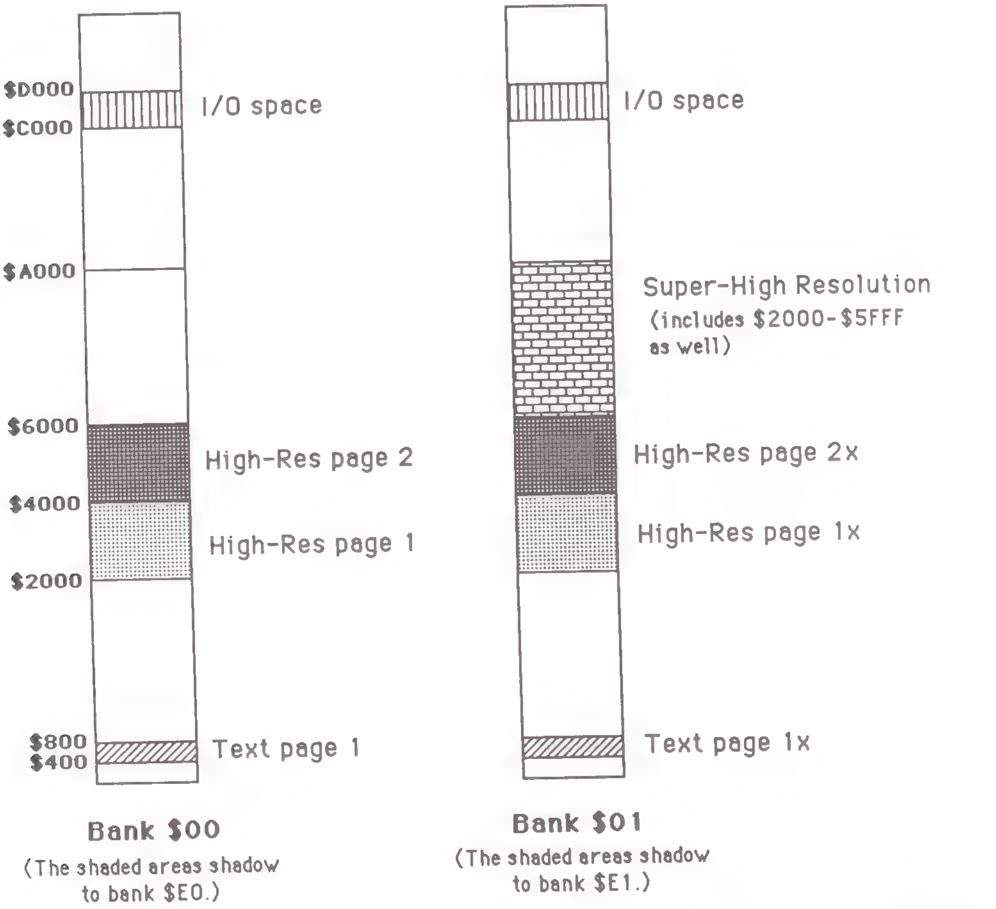
The ultimate solution was not to restrict the speed of bank $00 and $01 operations to 1 MHz. Rather, the system was designed to detect automatically a write operation to a video buffer or the I/O space (at a speed of 2.8 MHz or 1 MHz) and to generate a concurrent write operation (always at 1 MHz) to the same location in bank $E0 (for bank $00 writes) or bank $E1 (for bank $01 writes). This technique is called *shadowing*. Reading from a shadowed video buffer does not cause the system to slow down because neither bank $E0 nor bank $E1 is accessed.

The areas in banks $00 and $01 that may be shadowed are shown in figure 4–2. The GS has a *shadow register* that controls which video buffers are actually shadowed at any given time (see figure 4–3). A bit in the shadow register also controls the shadowing of I/O locations and language card operation.

Notice that the page 2 text and low-resolution graphics screen cannot be shadowed using hardware techniques, but it is rarely used anyway. (A software technique wherein data in bank $00/$01 is transferred to bank $E0/$E1 in response to a periodic interrupt can be implemented to emulate hardware shadowing. Invoke this by setting Alternate Display Mode On from the desk accessory menu.)

If shadowing is inhibited, a program must directly access the video buffers or I/O locations in bank $E0 or $E1. Enabling I/O shadowing also creates a language card

Figure 4–2. The Shadowed Memory Areas on the Apple IIGS

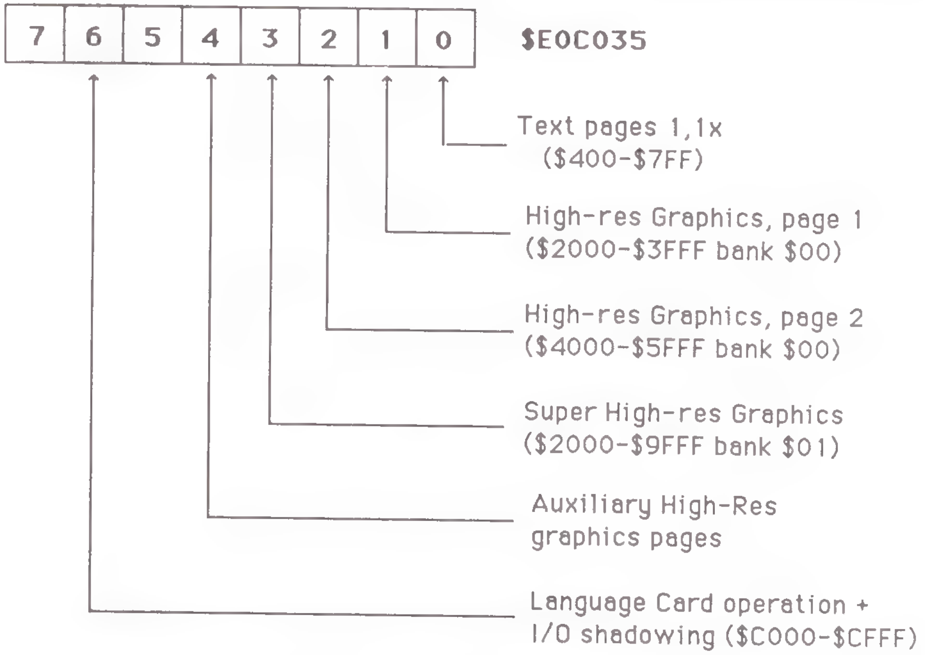


NOTE: Text page 2 ($800–$BFF) is shadowed using software techniques if Alternate Display Mode is On.

at $D000-$FFFF in banks $00/$01 that works just as the one in the IIe does (see the description above for banks $E0/$E1). This, in turn, enables access to the Applesoft and system monitor ROMs.

In normal Apple IIe emulation mode, all video areas are shadowed so that all existing software will work properly. When the GS's new super high-resolution graphics mode is used, shadowing of other graphics modes is disabled so that the shadowed space in banks $E0 can be freed.

Figure 4–3. The Shadow Register


7 6 5 4 3 2 1 0
$E0C035
Text pages 1,1x ($400-$7FF)
High-res Graphics, page 1 ($2000-$3FFF bank $00)
High-res Graphics, page 2 ($4000-$5FFF bank $00)
Super High-res Graphics ($2000-$9FFF bank $01)
Auxiliary High-Res graphics pages
Language Card operation + I/O shadowing ($C000-$CFFF)

1 = inhibit shadowing, 0 = enable shadowing

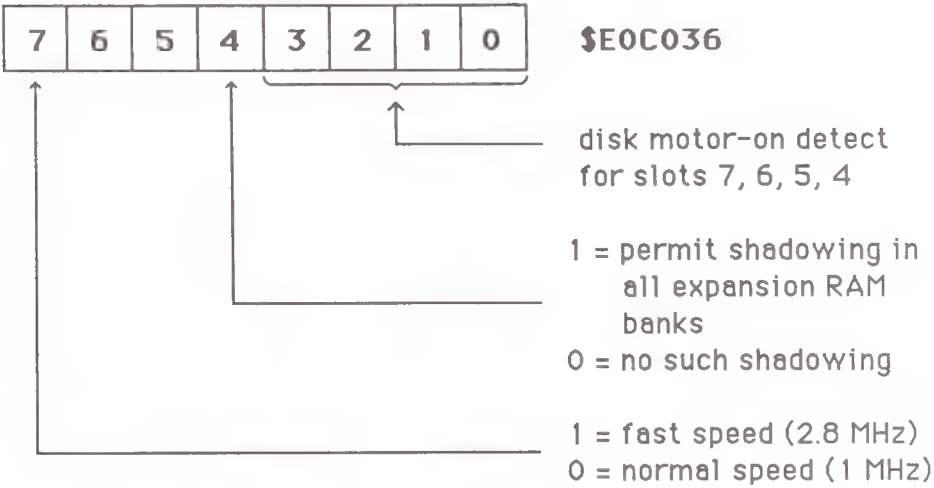
NOTE: Bit 4 works in conjunction with bits 1 and 2. If bit 1 is clear, the space from $2000–$3FFF in bank $00 is shadowed to bank $E0; if bit 4 is also clear, the same space in bank $01 is also shadowed (to bank $E1). Similarly, if bit 2 is clear, clearing bit 4 enables shadowing from $4000–$5FFF in bank $01 (to bank $E1) as well as the same locations in bank $00 (to bank $E0).

In normal operation, the GS always enables shadowing of the text pages, even if the text screen is not used. This is necessary so that IIe-style peripheral cards (which write to unused areas of the text display buffer) will work properly. Also, I/O shadowing is enabled so that interrupts (which vector through ROM in bank $00) can be handled.

EXPANSION RAM

Banks $02 through $DF (14 megabytes less 128K) are all reserved for future RAM expansion, although Apple has imposed a practical limit of 8 megabytes by not providing decoding circuitry beyond the 8-megabyte limit. Any expansion RAM you add is available for allocation with the Memory Manager.

Figure 4–4. The CYA ("Configure Your Apple") Register


7
6
5
4
3
2
1
0
$E0C036
disk motor-on detect for slots 7, 6, 5, 4
1 = permit shadowing in all expansion RAM banks
0 = no such shadowing
1 = fast speed (2.8 MHz)
0 = normal speed (1 MHz)

Video and I/O shadowing can be permitted for all of these banks as a group (but not individually) by setting a bit in the CYA register (see figure 4–4). Even-numbered banks shadow to page $E0 and odd-numbered banks to page $E1. However, you should never shadow expansion RAM if your are running GS-style applications because they will not run properly. The problems begin when the Memory Manager allocates space somewhere in the language card area of a bank, an area where ROM is usually enabled (just as it is in banks $00/$01). When a tool or a program tries to write to the allocated space, nothing is saved!

Refer to appendix 6 for descriptions of several memory expansion cards available for the GS.

THE ROM BANKS

The 128K ROM in banks $FE and $FF contains the ROM versions of many of the GS tool sets and desk accessory support code. It also contains, from $8000 to $FFFF in bank $FF, a system monitor comparable to that found on the IIe, the Applesoft BASIC programming language, and subroutines to support the built-in peripheral devices on the GS. By using clever hardware remapping techniques, the upper 12K of bank $FF can be used as if it occupied the same portion of bank $E0, $E1, or any other bank for which I/O shadowing is enabled.

The fourteen banks from $F0 to $FD are reserved for future ROM expansion. Banks $F0 through $F7 (512K) are to be used as a ROM disk and banks $F8 through $FD (384K) are for additional firmware ROM.

THE MEMORY MANAGER

The Memory Manager is tool set #2. The main chore it performs is allocating areas of memory in such a way that previously allocated areas are not disturbed. Its other major chore is releasing allocated areas of memory to the general pool of available memory.

The Memory Manager manages all of RAM memory except the following areas:

- $C000-$FFFF in banks $00, $01, $E0, $E1
- $0000-$07FF in banks $00, $01
- $0000-$1FFF in banks $E0, $E1

These unmanaged areas are reserved for use by the operating system.

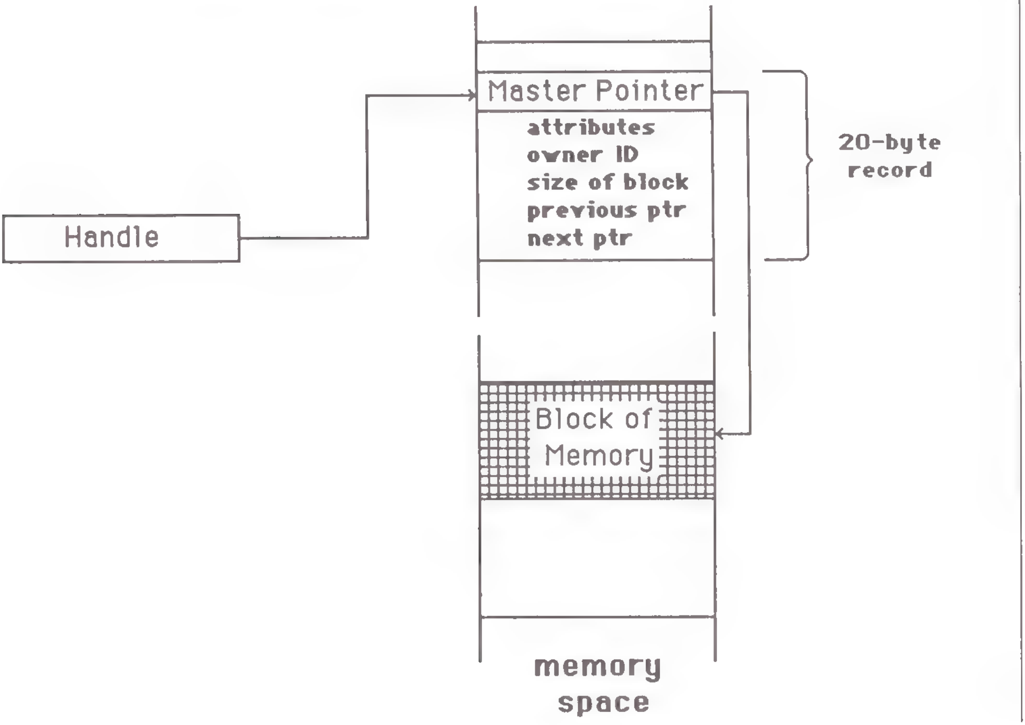
Areas of memory allocated by the Memory Manager are called *blocks*. Each block is identified by a four-byte address, called a *handle*, which is returned by the function performing the allocation. It is important to realize, however, that the handle is not the address of the block itself—instead, it is the address of a 20-byte block record. The first entry in this record is a pointer to the block (called a *master pointer*); it is the master pointer that contains the address of the block. See figure 4–5 for a pictorial representation of the relationship between a handle, a block record, a master pointer, and the block itself.

The advantage of using a handle instead of a pointer to identify a block is that a handle to a block remains valid even if the block is repositioned by the Memory Manager—only the address stored in the master pointer changes. If a pointer was returned instead, and the block was moved, a program would not be able to determine the address to which the block was moved.

The Memory Manager moves blocks around when it needs to perform a block-compaction operation. This occurs in only two situations: when you specifically request it (with CompactMem), or when you try to allocate a block (with NewHandle, ReallocHandle, or RestoreHandle) and there is no free area large enough to hold it. Block compaction gathers together allocated blocks which may be scattered throughout memory. This is done to eliminate gaps between blocks and to create as large a continuous free space as possible. Scattering occurs as blocks are allocated and deallocated during program execution.

If you are trying to allocate a block and there is no room for it even after compaction, the Memory Manager automatically frees purge-level-3 blocks (the Purge Level attribute is discussed below) and the blocks are compacted again. If

Figure 4–5. Block Allocation with the Memory Manager



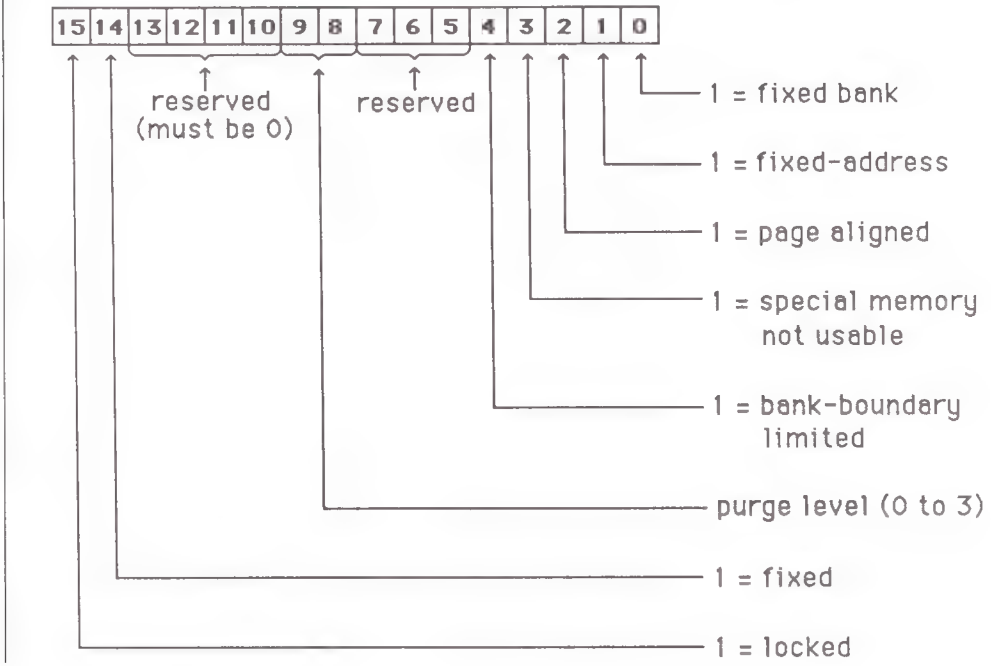
there still is not enough room, purge-level-2 blocks and then purge-level-1 blocks are freed; further compactions occur until there is room. An out-of-memory error is reported if there is no room even after all three levels of purgeable blocks have been purged.

You can tell the Memory Manager to free (or dispose of) previously allocated blocks by using the DisposeHandle function. It is good programming practice to dispose of blocks as soon as they are no longer needed so that as much space as possible is always available.

Attributes

When you make a block allocation request (with NewHandle or ReallocHandle), you must tell the Memory Manager the attributes of the block. These attributes indicate where the block is to be allocated and how it is to be dealt with after allocation.

Figure 4–6. The Attribute Word for a Block



The attributes are passed in a word; the bits in this word have the meanings shown in figure 4–6. Note that except for the Locked and Purge Level attributes, none of the attributes can be changed with function calls after the block has been allocated.

Locked. A locked block is one the Memory Manager is not permitted to move, even if the Fixed and Fixed Address attribute bits are off. In addition, a locked block cannot be purged even if its Purge Level is non-zero.

The Locked attribute can be changed after a block is allocated by calling HLock (to turn it on) or HUnLock (to turn it off). You should lock a block only when necessary, such as when you need to access the data it contains (see below), and then unlock it as soon as you can. Because locked blocks cannot be moved, they can prevent efficient block compaction operations and can lead to premature out-of-memory errors.

Fixed. Like a locked block, a fixed block may not be moved by the Memory Manager during a memory compaction operation; it stays put. The main difference between a fixed block and a locked block is that a fixed block can be purged if its Purge Level is non-zero. In addition, the setting of the Fixed attribute cannot be changed with a function call.

Purge Level. Purging a block means releasing a block by setting its master pointer to 0 (the space for the block record containing the master pointer is maintained and can be reused by calling ReallocHandle or RestoreHandle). The purged block returns to the pool of free memory that is available for future allocation by the Memory Manager, but the handle itself remains allocated and can be reused.

The Purge Level of a block can be 0, 1, 2, or 3, numbers that reflect the purge priority: blocks with higher purge levels are purged before blocks with lower purge levels. A Purge Level of 0 means that the block cannot be purged at all. The Purge Level of a block can be changed after allocation with the SetPurge function.

Purge level 3 is reserved for use by the GS System Loader, the tool set responsible for dealing with ProDOS 16 load files (which usually contain programs). When the System Loader loads a program, it first puts the previous program into a dormant (or "zombie") state by assigning a purge level of 3 to the memory blocks it uses. If any of these blocks are purged, they are all purged; if no purging takes place and control returns to the program, however, the System Loader does not reload the program from disk, it just sets the purge level of the block to zero and executes the program.

You can purge purgeable, unlocked memory blocks explicitly using PurgeHandle. Purging also occurs automatically if you call NewHandle, ReallocHandle, or RestoreHandle and there is not enough space for the new block even after the blocks are compacted. In this situation, purgeable blocks are purged, in priority order, until there is enough space.

Bank-Boundary Limited. If a block is Bank-Boundary Limited, it will be allocated in a single bank. Blocks that contain program code must have this attribute set, because 65816 programs cannot cross bank boundaries. (When the program counter goes from $FFFF to $0000 at a bank boundary, the carry generated is ignored and is *not* added to the program bank register. That means execution continues at the bottom of the current bank, not at the bottom of the next bank, as might be expected.)

Ordinary data blocks may span banks, however, because the 65816 indexed addressing modes let you index properly from one bank to the next.

Special Memory Not Usable. If this attribute is set, the block may not be allocated in bank $00 or $01, or in any of the graphics video buffer areas in banks $E0 or $E1 ($2000-$5FFF in $E0 and $2000-$9FFF in bank $E1). You would set this attribute if you were reserving memory from a IIe-style application so as to avoid allocating areas in the 128K program space.

Page-Aligned. If the block is Page-Aligned, it begins at an address of the form $bbxx00, that is, at the beginning of a memory page. (A memory page is the 256 bytes extending from address $bbxx00 to $bbxxFF.) You should set this attribute when allocating space to be used as a 65816 direct page so as to maximize the speed of direct page operations. A direct page operation still works if the page is not aligned, but it takes one cycle longer to execute than if it is aligned.

Fixed Address. Set this attribute if you want a block to be allocated at a specific location in memory. You might want to do this, for instance, to align a block with one of the graphics screen video buffers. The GS operating system uses this attribute when it reserves specific memory areas at start-up time.

Fixed Bank. If the Fixed Bank attribute is on, the Memory Manager will allocate a block in the bank specified when you call NewHandle, ReallocHandle, or RestoreHandle. You must set this attribute when allocating direct page and stack space, because the 65816's direct page and stack must be in bank $00.

Accessing a Block

To read data from or write data to a block, you need to know the address of the block. To get it, you must first *dereference* the handle to the block to determine what the master pointer is. Then, you can address the block by storing the master pointer in direct page and using the [dp],Y indirect addressing mode. (Recall from chapter 2 that the brackets indicate that this is a long addressing mode, not a short addressing mode; the pointer to the start of the block is stored at dp, dp+1, and dp+2, where dp is a direct page address.)

In the example to follow, a 256-byte block, whose handle is stored at MyHandle, is filled with zeroes. The first part of the subroutine dereferences the handle (DPScratch and ToBlock are the direct page addresses of two four-byte areas.):

```
; 16-bit A register
           LDA  MyHandle          ;Put handle in direct page
           STA  DPScratch         ; so that the master pointer
           LDA  MyHandle+2        ; can be accessed with an
           STA  DPScratch+2       ; indirect long addr mode.

           LDA  [DPScratch]       ;Get master pointer (low)
           STA  ToBlock
           LDY  #2
           LDA  [DPScratch],Y     ;Get master pointer (high)
           STA  ToBlock+2

           LDY  #0
           LDA  #0
```

```
ClearBlk  STA  [ToBlock],Y          ;Store a zero
          INY
          INY
          CPY  #256
          BNE  ClearBlk             ;Branch until Y=256
          RTS
```

An important caveat: You cannot use a dereferenced handle to access a block if the block may have been moved (because of compaction) or purged by the Memory Manager since the time you did the dereferencing—it may not point to the block anymore. Compaction and purging can occur if you call the Memory Manager directly, if you call a tool set function that calls the Memory Manager, or even if an interrupt occurs (the interrupt handler could call the Memory Manager).

To prevent the movement or purging of a block—and to maintain the validity of a dereferenced handle—lock the block in place with the HLock function just before you dereference. If you do this, be sure to unlock the block (with HUnLock) when you are through using the dereferenced handle. The alternative is to avoid calling tool set functions which may use the Memory Manager and to disable interrupts, restrictions which are usually impractical.

If you choose not to lock the block, you should dereference the handle to the block every time you want to access the block's data, but even this may not work if interrupts are occurring.

Of course, you do not have to worry about locking a block (or unlocking it) if you set the Fixed or Fixed Address attribute when the block was first allocated (and the block is not purgeable) or if you set the Locked attribute.

MEMORY MANAGER FUNCTIONS

There are many memory manager functions at your disposal, but most applications will use only a core group of six major functions. This chapter looks at this core group and then reviews most of the minor functions. (See table R4–1 at the end of this chapter for a summary of the major memory manager functions.)

The Major Functions

Start-up and Shut-Down. Like all tool sets, the Memory Manager has a start-up function, MMStartup, that your application must call before it can use the Memory Manager. Because almost every other tool set implicitly uses the Memory Manager, you must call MMStartup before calling the startup function of any other tool set except the Tool Locator and the Miscellaneous Tool Set.

MMStartup takes no input parameters, but it does return a result—an Owner ID tag. The ID tag is an identification code assigned to the program when the System Loader loads it into memory prior to execution. It is to be used to mark all blocks allocated by the application. It also identifies the blocks to be included in certain group operations the Memory Manager can perform.

Here is how to call MMStartup:

```
PHA                 ;Space for result (word)
_MMStartup
PLA
STA MyID            ;Save ID code
```

MyID is a two-byte data area that can be allocated with a data allocation directive of the form:

```
MyID    DS   2              ;Allocate two bytes
```

You should save MyID, because you will need it to start up other tool sets.

MMShutDown is the shut-down function for the Memory Manager. Call it just before the application ends, after shutting down all other tool sets (except the Tool Locator and the Miscellaneous Tool Set). MMShutDown has one input parameter (the ID tag returned by MMStartup) and returns no results:

```
PushWord MyID
_MMShutDown
```

Note that MMShutDown does not free up all memory blocks associated with MyID. This is done by ProDOS 16 when it regains control after a ProDOS 16 QUIT command. You can explicitly free up memory blocks with the DisposeHandle function (see below).

Block Allocation. NewHandle is the primary block allocation function. With it you can create blocks of any size, with any attributes, and you can associate them with any ID tag. NewHandle returns a handle to the allocated block.

The general calling sequence looks like this:

```
PHA                     ;Space for result (long)
PHA
PushLong BlockSize      ;Size of block
PushWord OwnerID        ;Owner ID tag
PushWord Attributes     ;Attributes of block
PushLong Location       ;Address of block
_NewHandle
PopLong MyHandle        ;Pop handle to block
```

(Recall from chapters 2 and 3 that PushWord, PushLong, and PopLong are macros, not 65816 instructions.)

BlockSize is the size of the block to be created, in bytes. OwnerID is the ID code to be assigned to the block and is normally the same as the ID code returned by MMStartup. Attributes is the attribute word for the block. Location is the starting address of the block and is meaningful only if the Fixed Address or Fixed Bank attribute is set.

NewHandle does all it can to reserve the requested space, including block compaction and purging, if necessary. The handle that is returned points to a block record made up of the following elements:

- Address of block (4 bytes)
- Block attributes (2 bytes)
- Owner ID tag (2 bytes)
- Size of block (4 bytes)
- Address of previous block record (4 bytes)
- Address of next block record (4 bytes)

If the record is the first or last one in the list maintained by the Memory Manager, the pointer to the previous block record or to the next block record is zero.

Notice that in the previous example, BlockSize, Attributes, and Location are passed as variables (numbers stored in memory locations), even though you are probably more likely to pass them as constants (immediate values). To pass constants, precede the constant with "#" to inform the PushLong or PushWord macro that it is a constant. For instance, to allocate five pages of memory in bank $00 for possible use as direct pages by other tools, call NewHandle like this:

```
PHA                     ;Space for result (long)
PHA
PushLong #$500          ;Five pages ($500 bytes)
PushWord MyID           ;ID returned by MMStartup
PushWord #$C005         ;Locked, Fixed, Fixed Bank, Aligned
PushLong #0             ;Address: bank $00
_NewHandle
PopLong MyHandle        ;Pop handle to block
```

Notice the attribute word for this block: $C005. Because a direct page must be in bank $00, the Fixed Bank attribute (bit 0) is set. Because 65816 direct page instructions work more quickly when direct page begins on a page boundary, the Page Aligned attribute (bit 2) is also set. The Locked attribute (bit 15) is set so that the block will not move around—this means that you will have to dereference its handle only once to access it. Finally, the Fixed attribute (bit 14) is set so that the block will not move even if it is accidentally unlocked.

Block Deallocation. As soon as you are finished with a previously allocated block of memory, you should free it up again to eliminate the possibility of an early out-of-memory error. To do this you can use the DisposeHandle function:

```
    PushLong MyHandle        ;Handle to block
    _DisposeHandle
```

DisposeHandle also removes the entry for the master pointer in the Memory Manager's list of master pointers. This means that you cannot reuse the handle with ReallocHandle or RestoreHandle.

DisposeHandle does its duties even if the block is locked or not purgeable.

Locking and Unlocking Blocks. As mentioned earlier, it is important to lock a moveable or purgeable block to ensure its dereferenced handle remains valid for data accesses. The function for doing this is HLock:

```
    PushLong MyHandle        ;Handle to block
    _HLock
```

The function for unlocking a block is _HUnLock. It also takes a handle to the block as its only parameter.

The Minor Functions

Most applications will not need to use the remaining Memory Manager functions very often—they are used primarily by the operating system. Nevertheless, it is important to become familiar with them so that you will recognize how to handle those situations calling for their use.

Reallocation of Purged Blocks. If a block has been purged, its master pointer is set to 0, but the space occupied by the master pointer is not freed. To reuse the master pointer of a purged block, call ReallocHandle. It behaves just as NewHandle does except that it requires one additional parameter, the handle to the purged block. This is the last parameter passed; the other parameters are pushed first in the same order as for NewHandle. You may find it simpler to use RestoreHandle. It needs only the handle; the file attributes, owner, and size previously used are retained.

When allocating space for a purged block, you should resist the temptation to use NewHandle. If you do not reuse purged handles, you will waste memory (20 bytes per handle, the size of the block record) and Memory Manager operations will be a bit slower.

Disposing and Purging Blocks. We saw earlier that DisposeHandle frees only the block associated with a particular handle. With DisposeAll, you can free up a group of blocks at once, each associated with the same ID tag. To use DisposeAll, pass the ID tag as a parameter:

```
   PushWord OwnerID          ;ID tag
   _DisposeAll
```

This example disposes of all blocks associated with OwnerID. Do not try to dispose of all blocks marked with the ID of the program itself, because the memory the program occupies could be overwritten before the program ends.

Another way to free up space is to purge a block or group of blocks with PurgeHandle or PurgeAll. Both functions will purge any unlocked block, even if the block's purge level is 0. The input parameter for PurgeHandle is a handle to the block to be purged. The parameter for PurgeAll is an ID tag—all blocks with the specified ID are purged.

Setting Attributes. Only the Locked and Purge Level attributes of a block can be modified after the block has been allocated. HLock and HUnLock, which lock and unlock individual blocks, have already been discussed. Two other functions, HLockAll and HUnLockAll, lock and unlock a group of blocks with the same ID tag. Pass the ID tag on the stack before calling either of these functions.

To set the purge level of a block, use SetPurge. It takes two parameters: the new purge level (a word from 0 to 3) and a handle to the block:

```
   PushWord #1               ;Purge Level = 1
   PushLong MyHandle         ;Handle to block
   _SetPurge
```

SetPurgeAll sets the purge level of every block with a given ID tag. The two parameters it requires are the new purge level and the ID tag.

Block Information Functions. FindHandle takes a memory location as an input parameter and returns a handle to the block in which it is located. For instance, a program can deduce the handle to the block in which it is currently running by executing the following segment of code:

```
           PHA                       ;Space for result (handle)
           PHA
           PushPtr  MyLabel          ;Push long address
   MyLabel _FindHandle
           PopLong CodeHndl          ;Pop the result (handle)
```

GetHandleSize takes a handle as an input parameter and returns the size of the block to which it refers. To get the size of the code segment in the above example, use the following code:

```
PHA                         ;Space for result (long)
PHA
PushLong CodeHndl           ;Handle to code block
_GetHandleSize
PopLong Size                ;Pop the result (long)
```

Use SetHandleSize to expand or contract the size of an existing block. The input parameters are a handle to the block and the new size of the block (a long word). For example, suppose your code block has an initialization sequence 743 bytes long which is located at the very end of the block. If you expect to be tight on memory space, you may want to dispose of it after you are through with it. Here is how you can do that:

```
PushLong CodeHndl           ;Push handle to code block
SEC
LDA     Size                ;New size = old size
SBC     #743                ; minus 743.
PHA
LDA     Size+2
SBC     #0
PHA
_SetHandleSize
```

Data Movement. The Memory Manager has four functions that permit you to move a block of data from anywhere in memory to anywhere else in memory. All you need is a pointer or handle to the start of the source block and to the destination block. If you use a handle, the Memory Manager function takes care of dereferencing it.

Here is a summary of how each of the four data movement functions work:

PtrtoHand Move a block referenced by a pointer to a block referenced by a handle

HandtoPtr Move a block referenced by a handle to a block referenced by a pointer

HandtoHand Move a block referenced by a handle to a block referenced by a handle

BlockMove Move a block referenced by a pointer to a block referenced by a pointer

Each function requires three long-word parameters on the stack: a pointer or handle to the source block, a pointer or handle to the destination block, and a count of the number of bytes to move.

Be very careful when using the data movement functions. They do not verify the validity of handles and pointers, and they do not check to see if the destination block is large enough to accommodate the transfer.

Free Space Functions. Three Memory Manager functions return numbers reflecting the sizes of various spaces in the system:

> FreeMem returns the number of free bytes in memory. The number does not include the space occupied by purgeable blocks. FreeMem does not compact memory.
>
> MaxBlock returns the size of the largest free block in memory. MaxBlock does not compact memory and it does not purge purgeable blocks. The number returned by MaxBlock cannot exceed the one returned by FreeMem because of block fragmentation.
>
> TotalMem returns the size of the RAM space in the system.

None of these functions takes input parameters, and all three functions return a long word (the size in bytes).

REFERENCE SECTION

Table R4–1: The Major Functions in the Memory Manager Tool Set ($02)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
BlockMove	$2B	SourcePtr (L)	Pointer to source block
		DestPtr (L)	Pointer to destination block
		Count (L)	Number of bytes to copy
CheckHandle	$1E	TheHandle (L)	Handle to be validated (*)
CompactMem	$1F	[no parameters]	
DisposeAll	$11	UserID (W)	ID tag for blocks to dispose
DisposeHandle	$10	TheHandle (L)	Handle to block to dispose
FindHandle	$1A	result (L)	Handle for block
		MemLocation (L)	Location to test
FreeMem	$1B	result (L)	Amount of free memory

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
GetHandleSize	$18	result (L)	Size of block in bytes
		TheHandle (L)	Handle to block
HandtoHand	$2A	SourceHndl (L)	Handle to source block
		DestHndl (L)	Handle to destination block
		Count (L)	Number of bytes to copy
HandtoPtr	$29	SourceHndl (L)	Handle to source block
		DestPtr (L)	Pointer to destination block
		Count (L)	Number of bytes to copy
HLock	$20	TheHandle (L)	Handle to block to lock
HLockAll	$21	UserID (W)	ID tag of blocks to lock
HUnLock	$22	TheHandle (L)	Handle to block to unlock
HUnLockAll	$23	UserID (W)	ID tag for blocks to unlock
MaxBlock	$1C	result (L)	Size of largest free block
MMShutDown	$03	UserID (W)	ID tag returned by MMStartup
MMStartup	$02	result (W)	ID tag for memory allocation
NewHandle	$09	result (L)	Handle to new memory block
		BlockSize (L)	Size of memory block
		UserID (W)	ID tag for memory block
		MemAttrib (W)	Attributes of memory block
		MemLocation (L)	Allocation address
PtrtoHand	$28	SourcePtr (L)	Pointer to source block
		DestHndl (L)	Handle to destination block
		Count (L)	Number of bytes to copy

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
PurgeAll	$13	UserID (W)	ID tag for blocks to purge
PurgeHandle	$12	TheHandle (L)	Handle to purge
ReallocHandle	$0A	BlockSize (L)	Size of memory block
		UserID (W)	ID tag for memory block
		MemAttrib (W)	Attributes of memory block
		MemLocation (L)	Allocation address
		TheHandle (L)	Handle to re-use
RestoreHandle	$0B	TheHandle (L)	Handle to be restored
SetHandleSize	$19	NewSize (L)	New size of block
		TheHandle (L)	Handle to block to be resized
SetPurge	$24	PurgeLevel (W)	New purge level
		TheHandle (L)	Handle to memory block
SetPurgeAll	$25	PurgeLevel (W)	New purge level for blocks
		UserID (W)	ID tag for blocks involved
TotalMem	$1D	result (L)	System memory size

* For CheckHandle, an error code of $0206 is returned if the handle is not valid.

Table R4–2: Memory Manager Error Codes

Error Code	*Description of Error Condition*
$0201	Out of memory; the block was not allocated.
$0202	The operation failed because the block has been purged.

Error Code	*Description of Error Condition*
$0203	An attempt was made to reallocate (with ReallocHandle) a handle of a block that has not been purged yet.
$0204	The operation failed because the block is locked or immovable.
$0205	An attempt was made to purge an unpurgeable block.
$0206	The handle does not exist.
$0207	The ID tag does not exist.
$0208	Illegal operation on a block with the specified attributes.

CHAPTER 5

Event Management

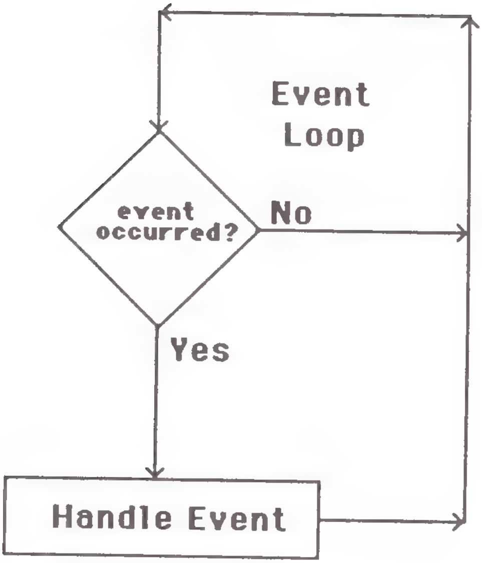
An important part of any application program is the code that checks for user input and responds to it in an appropriate way. The two main sources of user input on the GS are the keyboard and the mouse, but incoming data can also come from any other device on the Apple DeskTop Bus or from a device connected to a port or slot. The occurrence of a discrete element of input activity is called an *event* and the tool set an application uses to monitor events is called the Event Manager.

The Event Manager actually handles more than just traditional mouse and keyboard events. It also manages two window-related operations that take place when the appearance of a window on the graphics screen is to be changed—namely, activate/deactivate events and update events. (Windows will be discussed in detail in the next chapter.) The GS operating system treats activate/deactivate and update activities as events to make it easier to develop software working in a multiwindow environment.

THE EVENT LOOP

Any GS application that uses the desktop environment should be event-driven. That is, the application should be ready, at all times, to react to any event which might occur. It should avoid putting the user in a mode that restricts the group of actions the user may perform, because this only serves to frustrate the user. Of course, there are times when restrictive modes are desirable, such as when the application wants to force the user to respond to an important question before proceeding further, but such modes are not needed often.

An event-driven application spends most of its time in an event loop, in which it repeatedly checks for the occurrence of an event. When an event occurs, the application responds to it and then returns to the loop and waits for the next event. Here is the simplified flowchart of an event-driven application:



As you can see, the main task of an event loop is to check whether an event has occurred. This is a very simple task involving the use of the Event Manager's GetNextEvent function. The actual handling of an event is more complicated but still quite straightforward. The most difficult part of any application, of course, is the part actually performing the tasks selected by the user.

Initializing the Event Manager

Chapter 3 explained that before you can use any tool set you must initialize it by calling its start-up function. The name of the start-up function for the Event Manager is EMStartup. Call it right after starting up the Tool Locator and the Memory Manager with calls to TLStartup and MMStartup.

The calling sequence for EMStartup looks like this:

```
PushWord DPAddr          ;One page in bank $00
PushWord #QueueSize      ;Size of event queue
PushWord #XMinClamp      ;Mouse minimum (horiz)
PushWord #XMaxClamp      ;Mouse maximum (horiz)
```

```
    PushWord #YMinClamp      ;Mouse minimum (vert)
    PushWord #YMaxClamp      ;Mouse maximum (vert)
    PushWord UserID          ;ID returned by MMStartup
    _EMStartup
```

EMStartup does not return a result.

DPAddr is the starting address of a 256-byte area in bank $00 that the Event Manager uses as a direct page. EMStartup puts this address in the Work Area Pointer table using the Tool Locator's SetWAP function. Use the Memory Manager's NewHandle call to allocate the bank $00 space:

```
    PHA                      ;Space for result
    PHA
    PushLong #256            ;Need one page (256 bytes)
    PushWord UserID          ;ID returned by MMStartup
    PushWord #$C005          ;Locked, Static, Fixed Bank, Aligned
    PushLong #$000000        ;Address: bank $00
    _NewHandle
    PopLong MyHandle         ;Pop handle to block
```

To determine the actual direct page address to be used as DPAddr, you must first dereference the handle using the techniques described in the last chapter. The address is the low-order word of the dereferenced handle.

QueueSize is the number of events the Event Manager can keep track of at one time. The events are posted in a queue as they are detected and are removed one at a time as the application calls GetNextEvent. If the application does not retrieve events from the queue often enough, the queue could fill up, causing older events to be lost as they are replaced by more recent events. QueueSize must be an integer between 0 and 3639. If a value of 0 is specified, a default size of 20 is used; this is a suitable size for most applications.

The four clamp parameters describe the coordinates of the rectangle in which the mouse pointer will be confined on the screen. XMinClamp and XMaxClamp refer to the left and right sides of the rectangle, and YMinClamp and YMaxClamp refer to the top and bottom sides. The clamp parameters are usually set to the coordinates of the full graphics screen so that the user can point to any object on the screen. This means that XMinClamp and YMinClamp will be 0, and YMaxClamp will be 200. The value of XMaxClamp depends on the graphics mode in use; it will be 320 for 320-by-200 mode, or 640 for 640-by-200 mode. To ensure that you can always see the mouse's arrow cursor, however, you may want to set XMaxClamp and YMaxClamp to values that are two or three pixels smaller than the full screen dimensions.

UserID is the ID code returned by MMStartup. It is used when the Event Manager calls the Memory Manager to allocate any blocks it needs to operate.

To shut down the Event Manager, call EMShutdown. It requires no parameters.

Table 5–1: Events Handled by the Event Manager

Mouse Events	*Keyboard Events*
Mouse-down	Key-down
Mouse-up	Auto-key
Window Events	*Special Events*
Update	Desk accessory
Activate	Switch
	Device driver
	Application-defined (4)

EVENT TYPES

The Event Manager supports 16 types of events. This includes a null event that is reported when no other type of event has taken place before a request for an event is made with GetNextEvent. Two of the 16 event types are not currently in use, so there are really only 14 event types that may be reported.

The different events handled by the Event Manager can be broken into four groups: mouse events, keyboard events, window events, and special events. See the breakdown in table 5–1.

Mouse Events

There are two mouse events, mouse-down and mouse-up. A mouse-down event occurs when a released mouse button is pressed. A mouse-up event occurs when a pressed mouse button is released.

The Event Manager actually works with a two-button mouse (or an alternative pointing device with two buttons), but the same event codes for mouse-up and mouse-down are reported for either button. You can determine which button was actually involved in the event by examining the Modifiers field of the event record returned by GetNextEvent, as you will see later in this chapter.

You can determine the state of the mouse button directly with the Button function:

```
PHA             ;space for result
PushWord #0     ;button number (0 or 1)
_Button
PLA             ;True = button is down
```

For the standard GS mouse, the button number must always be zero. The Boolean result is true (non-zero) if the button is down or false (zero) if it is not.

Keyboard Events

The two keyboard events are key-down and auto-key. A key-down event occurs when a released character key is pressed. A character key is one which generates an ASCII character code when pressed by itself. Every key on the keyboard is a character key except the Shift, Caps Lock, Open-Apple (Command), Control, and Option (Solid-Apple) keys. These keys are called *modifier keys* because they are usually pressed at the same time as a character key to change, or modify, the ASCII code it would otherwise generate.

If you hold down a character key long enough it will begin to repeat itself. The act of repetition is called an auto-key event. The repeat rate can be set with the Control Panel desk accessory; you can even turn off the auto-repeat feature if you wish.

Window Events

The two window-related events are update and activate. An update event occurs when a portion of a window on the screen becomes exposed to view. This happens when the window is first created, when an overlapping window is moved aside, or when the window is enlarged.

An activate event occurs when a window becomes active or inactive. By convention, an active window is one that is to be affected by subsequent commands and drawing operations. The active window is highlighted in a distinctive way on the screen.

To determine the reason for an activate event, you can examine a bit flag in the Modifiers flag of the event record returned by GetNextEvent. If the bit is 0, the activate event actually corresponds to a window becoming inactive.

Special Events

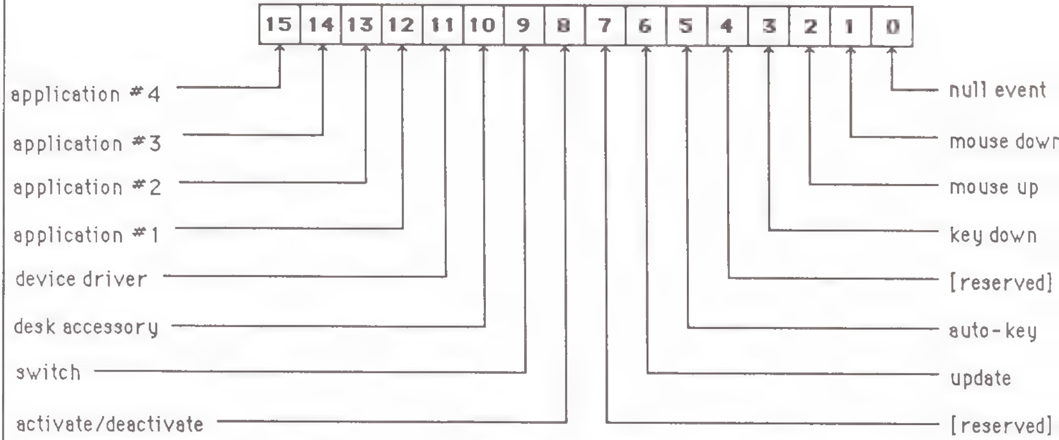
Most of the events not already discussed above are rarely used by an application. There are four application-defined events an application can post in the queue to report a custom event, perhaps the occurrence of a specific combination of events.

The desk accessory event occurs when a user enters Control-OpenApple-Esc from the keyboard. It is never reported to the application by GetNextEvent; rather, it is trapped by the operating system's main event-handling routine and handled by popping up the Classic Desk Accessory menu (the one containing the Control Panel entry).

A switch event occurs when the user clicks a switcher control icon to transfer control to another application in memory. A switcher program is not yet available for the GS, however.

A driver for an I/O device can report a device driver event when it has received data. It may also report an event if it is ready to send data.

Figure 5–1. The Event Mask Used by GetNextEvent and TaskMaster



NOTE: An event type is selected if its bit is 1; otherwise it is ignored.

GETNEXTEVENT AND THE EVENT LOOP

The Event Manager function that scans the event queue for the presence of an event is GetNextEvent. GetNextEvent returns a Boolean (true/false) result indicating whether there was anything in the queue. If the queue is empty, the result is false (zero), and the application can complete the event loop by calling GetNextEvent again.

Here is how to call GetNextEvent:

```
PHA                         ;space for result
PushWord #$FFFF             ;event mask
PushPtr EventRec            ;Pointer to event record
_GetNextEvent
PLA                         ;Pop Boolean result
```

The event mask tells GetNextEvent which events are to be reported. As shown in figure 5–1, each bit in the event mask word corresponds to a particular type of event. If the bit corresponding to an event is 1, that event can be reported. If you are interested in all types of events, pass an event mask of $FFFF to GetNextEvent.

EventRec is a 16-byte event record. GetNextEvent fills the event record with information describing the event it removes from the queue. The structure of an event record is as follows:

```
EventRec    ANOP
what        DS    2       ;event code (integer)
message     DS    4       ;event message (long word)
when        DS    4       ;ticks since system startup (long word)
where       DS    4       ;mouse position (point)
modifiers   DS    2       ;modifier flags (integer)
```

It is also advisable to reserve space for two other long word fields immediately following the event record: TaskData and TaskMask. They are used by the TaskMaster function (described below), not by GetNextEvent. You will probably use TaskMaster more often.

The What Field

The What field of the record contains the event code, an integer from 0 to 15 identifying what type of event occurred:

0 = null event (no event occurred)
1 = mouse-down event
2 = mouse-up event
3 = key-down event
4 = [not used]
5 = auto-key event (a key was repeated)
6 = window update event
7 = [not used]
8 = window activate event
9 = switch event
10 = desk accessory event
11 = device driver event
12 = application-defined event
13 = application-defined event
14 = application-defined event
15 = application-defined event

Notice that two codes in this range, 4 and 7, are not presently defined. On the Macintosh, an event code of 4 corresponds to a key-up event.

The Message Field

The Message field of the event record holds a 4-byte *event message* containing more information about the event. The content of the event message depends on the event type:

mouse-down	button number (0 or 1)
mouse-up	button number (0 or 1)
key-down	ASCII character code (0 to 127)
auto-key	ASCII character code (0 to 127)
window update	Pointer to window
window activate	Pointer to window
device driver	Defined by the device driver
application-defined	Defined by the application

The event message for all other event types is undefined.

The When Field

The When field contains the time when the event took place. It is expressed in *ticks* since system start-up. A tick is a unit of time equal to one-sixtieth of a second.

The Where Field

The Where field holds the vertical and horizontal position of the mouse on the screen when the event occurred. The position is calculated using the *global coordinate system.* In this coordinate system, the origin is the top left-hand corner of the screen and the horizontal and vertical axes increase to the right and down, respectively.

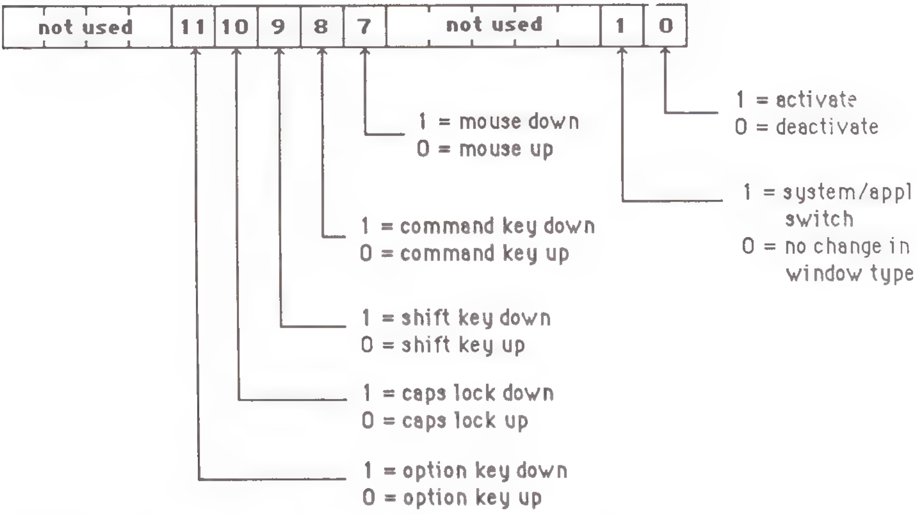
The Modifiers Field

The last field in an event record holds the *modifier flags* (see figure 5–2). These flags indicate the states of the keyboard modifier keys, the keypad, 'and the mouse buttons when the event took place.

Modifier keys are keys that, by themselves, do not generate character codes when you press them: Shift, Caps Lock, Control, Option (also called Solid-Apple), and Open-Apple (also called Command). An application will inspect the modifier flags if it is designed to differentiate between events that take place when a modifier key is pressed and events that are unmodified. For example, an application might want to let a user select a single icon by clicking a mouse button over that icon, or it might allow the user to select a group of icons by clicking the mouse button over each icon while holding down either Shift key. When such an application receives a mouse-down event from GetNextEvent, it would check the modifiers field to see if the Shift key was being pressed.

The flag for a modifier key is 1 if the key is down, and 0 if it is up. The Btn0State and Btn1State flags behave in the opposite way—a 1 corresponds to a button-up state.

Figure 5–2. The Format of the Modifiers Field of an Event Record or Task Record



Command Key = Open-Apple Option Key = Solid-Apple

Two other flags in the Modifiers field relate only to window activate/deactivate events. One flag, ActiveFlag, indicates whether the window is being activated (1) or deactivated (0), and the other flag indicates whether the newly activated window is of a different type from the previously active window (1) or whether it is the same type (0). (Windows are either application windows or system windows. System windows are those created by New Desk Accessories; see chapter 9.)

The Event Loop

The code for a simple event loop is remarkably straightforward:

```
EvtLoop PHA                ;Space for result
        PushWord #$FFFF    ;Event mask: all events
        PushPtr EventRec   ;Pointer to event record
        _GetNextEvent
        PLA                ;Get the Boolean result
        BEQ EvtLoop        ;Branch if no event
```

Until an event occurs, the code between EvtLoop and the "BEQ EvtLoop" instruction is executed again and again. This is because the result is always false ($0000), which means that the zero flag is set after the PLA instruction. When an event is reported, the result is true (non-zero), the zero flag is cleared by the PLA instruction, and control passes through the bottom of the loop.

The code following the loop can read the event type code from the What field of the event record and then pass control to the subroutine that handles that event type. Here is an example of a piece of code that does just that:

```
          LDA   What             ;Get event type code
          ASL   A                ;Double it to get table offset
          TAX                    ;Set up index for JMP (addr,X)
          JMP   (EvtTable,X)     ;Jump to event handler

EvtTable  DC    I'Ignore'        ;Null event
          DC    I'DoMouseDown'   ;Mouse-down
          DC    I'DoMouseUp'     ;Mouse-up
          DC    I'DoKeyDown'     ;Key-down
          DC    I'Ignore'        ;[not used]
          DC    I'DoMouseDown'   ;Auto-key
          DC    I'DoUpdate'      ;Window update
          DC    I'Ignore'        ;[not used]
          DC    I'DoActivate'    ;Window activate
          DC    I'Ignore'        ;Switch
          DC    I'Ignore'        ;Desk Accessory
          DC    I'DoDriver'      ;Device Driver
          DC    I'DoAppl1'       ;Application-defined #1
          DC    I'DoAppl2'       ;Application-defined #2
          DC    I'DoAppl3'       ;Application-defined #3
          DC    I'DoAppl4'       ;Application-defined #4

Ignore   JMP   EvtLoop
```

Notice how this technique works. The entries in EvtTable are the addresses of the handlers for each event type, in event-type order. Because each address is two bytes long, the entry for a particular event is at an offset from EvtTable equal to two times the event type code value. This means that you can jump to the handler with a JMP (EvtTable,X) instruction, where X holds the offset.

If the application deals with only a few types of events, it may be more convenient to use a series of CMP (compare) instructions to route the application to the proper event handler. Suppose, for example, an application handles mouse-down and key-down events but ignores every other type of event. You could pass control to the two event handlers using a code fragment like this:

```
         LDA What              ;Get event type code


         CMP #1                ;Is it mouse-down?
         BNE CheckKD           ;No, so branch
         BRL DoMouseDown       ;Yes, so branch
```

```
CheckKD CMP #3                     ;Is it key-down?
        BNE Ignore                 ;No, so branch
        BRL DoKeyDown              ;Yes, so branch

Ignore  JMP EvtLoop                ;Back to the event loop
```

Notice that BRL rather than BRA instructions are used to branch to the event-handling subroutines. This is because the target address of a BRL instruction can be anywhere in the 64K program bank, whereas the target address for a BRA instruction is restricted to an address in the range -128 to +127 bytes from the next instruction.

If GetNextEvent returns a non-null event, it automatically removes the event from the queue. To determine what the next event is without removing the event from the queue, use EventAvail:

```
PHA                        ;space for result
PushWord #$FFFF            ;event mask
PushWord EventRec          ;Pointer to event record
_EventAvail
PLA                        ;Boolean: always true
```

After calling EventAvail, information about the next event in the queue can be found in the event record.

Before entering an event loop at the beginning of a program, you may want to remove any events which may be lingering in the event queue. To do this, call FlushEvents:

```
PHA                       ;space for result
PushWord #$FFFF           ;EventMask for events to remove
PushWord #$0000           ;StopMask for events to keep
_FlushEvents
PLA                       ;Mask for first event not removed
```

The first mask passed to FlushEvents describes the types of events that are to be removed from the queue. The second mask, called StopMask, describes the types of events that are to stop the removal process. FlushEvents scans the queue from the beginning and removes all events up to (but not including) any event described by StopMask. The result it returns is the mask for the first event which was not removed. If the result is zero, all events were removed.

In most cases you will want to remove every event from the queue. To do this, use an EventMask of $FFFF and a StopMask of $0000, as in the example.

POSTING EVENTS

In certain situations it may be convenient to place events in the event queue yourself. For example, you would do this to pass application-defined events to the system. To place an event in the queue, call PostEvent:

```
PHA                           ;space for result
PushWord EventCode            ;The event code
PushLong EventMsg             ;The event message
_PostEvent
PLA                           ;True = event was posted
```

PostEvent places the event described by EventCode and EventMsg in the event queue. It automatically assigns the current time, mouse location, and state of the modifier keys and mouse button to the other fields of the event record. There is an exception: for keyboard or mouse events, the modifiers word is set to the high-order word of EventMsg.

HANDLING EVENTS

An application may handle an event any way it wants to, of course. There are, however, certain rules that every application should follow when handling events so as to conform to Apple's user-interface guidelines. This section explains what these rules are and how to react in special ways to mouse, keyboard, and window events.

Before proceeding, a comment should be made about a useful function that is actually part of the Window Manager, not the Event Manager: TaskMaster. You use it much as you would use GetNextEvent; the difference is that TaskMaster processes certain events in accordance with the user-interface guidelines before returning control to the caller. This allows the programmer of an application to concentrate on writing code unique to the application.

Here is an event loop using TaskMaster instead of GetNextEvent:

```
EvtLoop PHA                              ;space for result
        PushWord #$FFFF                  ;Event mask
        PushPtr TaskRec                  ;Pointer to task record
        _TaskMaster
        PLA                              ;Pop the task code
        BEQ        EvtLoop               ;Branch if nothing to do
```

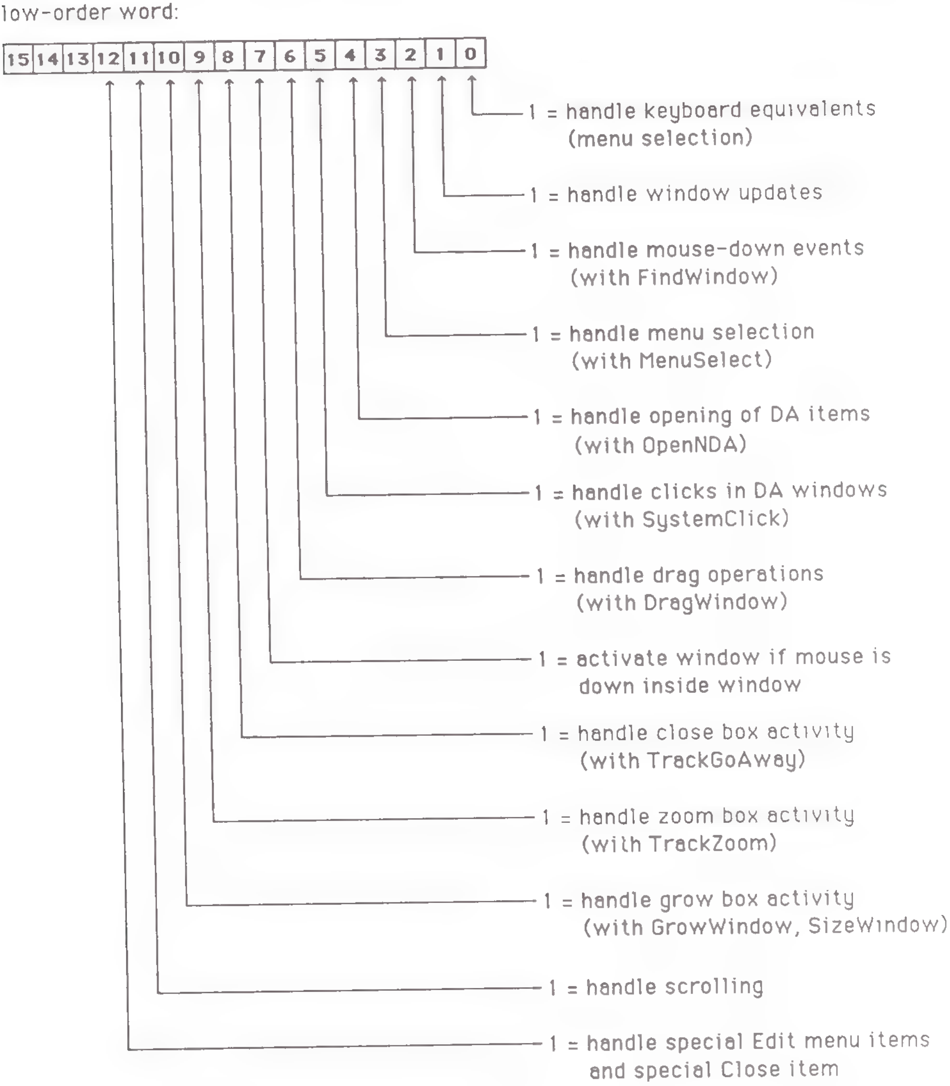
Looks familiar, right?

The program in listing 5–1 is a simple program shell that might assist you in developing more complex desktop applications. At its core is a simple TaskMaster event loop.

The task record referred to in this calling sequence is just a standard event record followed by long word TaskData and TaskMask fields.

TaskData is a long word in which TaskMaster sometimes returns a result (usually the pointer to a window). TaskMask is a bit vector indicating just which tasks TaskMaster is to handle (see figure 5–3). If the bit for a particular task is 0, the event code for the task is returned just as if you had called GetNextEvent. It is then up to the application to deal with it in an appropriate way. In most cases, you

Figure 5–3. The Format of the Task Mask Used by TaskMaster


low-order word:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 = handle keyboard equivalents (menu selection)
1 = handle window updates
1 = handle mouse-down events (with FindWindow)
1 = handle menu selection (with MenuSelect)
1 = handle opening of DA items (with OpenNDA)
1 = handle clicks in DA windows (with SystemClick)
1 = handle drag operations (with DragWindow)
1 = activate window if mouse is down inside window
1 = handle close box activity (with TrackGoAway)
1 = handle zoom box activity (with TrackZoom)
1 = handle grow box activity (with GrowWindow, SizeWindow)
1 = handle scrolling
1 = handle special Edit menu items and special Close item

NOTE: Bits 13 through 31 of the task mask must be zero.

will want TaskMaster to handle everything it is designed for, so set the mask to $00001FFF.

The first thing TaskMaster does is call GetNextEvent to get an event with which it can work. If then deals with the event in a standard way and returns a task code; if the code is 0, there is nothing more for the application to do and it can loop back and call TaskMaster again.

Assuming the standard task mask is used, TaskMaster may return the following codes:

`inKey` (3) When a key is pressed which is not the keyboard equivalent of a menu item

`inUpdate` (6) When the window to be updated does not have an update drawing subroutine; see chapter 6.

`wInMenuBar` (17) When a menu item is selected which is not a desk accessory item or a special item (undo, cut, copy, paste, clear, close); see chapter 7.

`wInContent` (19) When a mouse-down event occurs in the content region of the active window

`wInGoAway` (22) When a mouse-down event occurs in a close box of a window and the button is released in the close box.

`wInSpecial` (25) When a special menu item is selected which is not accepted by a desk accessory

`wInFrame` (27) When a mouse-down event occurs in the frame of an active window (other than scroll controls)

TaskMaster also passes through mouse-up, auto-key, device driver, and application-defined events with no pre-processing of any kind.

The sections that follow cover ways to handle standard Event Manager events. In general, the descriptions given simply describe how TaskMaster handles the events. There are no special requirements for reacting to switch, device driver, and application-defined events. In addition, the desk accessory event is handled internally by the Event Manager.

Mouse-down

When a mouse-down event occurs, the user is usually selecting an object on the screen. The program must determine what the object is—it could be a menu bar, a window, a scroll control, or a line of text, for example—and then process the selection in an appropriate way before returning to the main event loop. Techniques for handling mouse-down events will be described in detail in chapters 6 and 7.

Mouse-up

Mouse-up events are often handled by terminating an activity that began with the previous mouse-down event. The classic example of such an activity is the dragging of an object on the screen: while the mouse button is down, a selected object follows the cursor around the screen; when the button is released, the object is released and fixed in place. The program in listing 5–2, for example, lets you size a rectangle by dragging the mouse and letting go.

Mouse-up events may also be handled by determining whether the button was released when the mouse cursor was still in the same position on the screen as when the mouse was first pressed. If it was not, the application may want to ignore the mouse click. For example, the user-interface guidelines insist that a program ignore a mouse-down event in the close or zoom box of a window if the button is not also released in the box. The rationale for this dictum is that if the user leaves the vicinity of the box, he probably wants to cancel the operation.

Another operation involving a mouse-up event is a double-click—the pressing and releasing of the mouse button twice in quick succession. A double-click is frequently used to shortcut an operation that normally requires two separate clicks, such as the launching of an application from a program selector utility. Two consecutive clicks are to be treated as a double-click, instead of as two unrelated clicks, if the following conditions are true: the time (in ticks) between the mouse-up event for the first click and the mouse-down event for the second must be less than the DblTime parameter; and both clicks must take place within the same object or within a few pixels of the object.

To read the current setting of the DblTime parameter, use the Event Manager's GetDblTime function:

```
PHA                 ;Space for result (long word)
PHA
_GetDblTime
PLA                 ;Pop the result
STA DblTime         ; (low)
PLA
STA DblTime+2       ; (high)
```

The default value for DblTime is 30 ticks, but it can be adjusted using the GS's Control Panel. The program in listing 5–3 shows how to detect double clicks; when it finds one, it draws a small black box on the screen.

In each of the above three scenarios, the mouse-up event is closely related to the previous mouse-down event. For this reason, mouse-up events are best handled during the processing of the mouse-down event. Mouse-down events that occur in the main event loop can be ignored.

In the handler for a mouse-down event, use the Event Manager's StillDown function to wait for a mouse-up event to occur so that you can stop dragging an object, check that the mouse is still in an action box, check for a double-click, or whatever. Until the StillDown function returns a false (zero) result, the mouse button is still down. Here is the calling sequence for StillDown:

```
PHA                 ;space for Boolean result
PushWord #0         ;button number (always 0 for mouse)
_StillDown
PLA                 ;Pop the result
BNE ItsDown         ;Branch if it is still down
BEQ ItsUp           ;Branch if it is up
```

StillDown does not actually remove the mouse-up event from the event queue—GetNextEvent takes care of that when you return to the event loop. If you prefer that the event be removed, use the WaitMouseUp function instead of StillDown; in all other respects the two functions work the same. You might want to use WaitMouseUp if your event loop does not ignore mouse-up events.

Key-down and Auto-key

An application should react to a key-down event by checking to see if it corresponds to the keyboard equivalent of a menu item (this is covered in chapter 7). If it is not, there is no standard way of dealing with the character—it depends on the application. If the user is entering a line of text, for example, the program should display the character on the screen and update the active cursor position.

Auto-key events are usually handled in the same way as key-down events. You may want to ignore them if they correspond to menu item equivalents, however, because you do not usually want commands entered from menus to repeat automatically.

Window Update

An update event occurs when a portion of a window needs to be redrawn. Such an event occurs in the following situations:

- When the window first appears on the screen (after NewWindow or ShowWindow)
- When an overlapping window is moved aside to make visible a new portion of the window's content region
- When the window is enlarged by tugging on its grow box
- When a portion of the content region is made invalid with InvalRect or InvalRgn

The window's update region defines the portion of a window that requires redrawing. This may be the entire content region or any part of it. Details on how to handle update events will be given in chapter 6.

Window Activate

Window activate events actually include deactivate events. To distinguish between the two, examine bit 0 of the Modifiers field of the event record. If it is 1, it is an activate event.

Activate events are handled by highlighting the window and bringing it to the front of the screen. Highlighting involves drawing racing stripes in the title portion of the window and drawing the detailed structure of scroll controls. These areas in an inactive window are white.

Activate events are described in more detail in chapter 6.

CURSORS

The cursor-handling functions are actually part of the QuickDraw tool set (see chapter 6) but it is appropriate to discuss them here because they are tied to a rather common user activity (although not an event handled by the Event Manager): moving the mouse. A cursor is a small icon that moves around the screen as you move the mouse around the tabletop. It serves to inform the user of the position of the mouse on the screen.

(A program can determine the current position of the mouse by calling GetMouse:

```
             PushPtr MousePosn  ;Ptr to space for result
             _GetMouse
             RTS
MousePosn    DS 4               ;Position in local coordinates
```

Note that the mouse position is returned in the local coordinates of the currently active drawing window. See the next chapter for a discussion of local coordinates.)

The standard cursor is a small arrow. To make it visible, call InitCursor (no parameters) after starting up QuickDraw. You can switch to a custom cursor using the SetCursor function:

```
PushPtr TheCursor          ;Pointer to cursor record
_SetCursor
```

The Cursor points to a cursor record that defines the size and shape of the cursor. The structure of this record is as follows:

Cursor height (word)	Height of cursor rectangle in rows
Cursor width (word)	Width of cursor rectangle in words
Cursor image (bytes)	Cursor definition, row by row
Cursor mask (bytes)	Cursor mask, row by row
Hot spot Y (word); Hot spot X (word)	The hot spot is the position in the rectangle that is aligned with the mouse position

Note that the width of the cursor takes into account the "chunkiness" of the graphics screen, that is, whether two (640-by-200 mode) or four (320-by-200 mode) bits are needed to define a dot on the screen. (Screen resolution is discussed in detail in the next chapter.) Note also that the last word in the group of words defining a row in the cursor must be zero.

The cursor mask indicates which parts of the cursor image are to be drawn on the screen. Only those bits in the cursor image that correspond to 1 bits in the cursor mask are used. The cursor mask is generally a filled-in outline of the cursor surrounded by a one-pixel "glove." This makes the cursor visible even if it is positioned over an area that is the same color as the cursor.

Below is the record for a cursor in 320 graphics mode (4 bits per pixel); this forms an I-beam, the standard text-insertion cursor.

```
DC      I2'11'                          ;Height (in pixel rows)
DC      I2'4'                           ;Width (in words)


DC      H'0000000000000000'
DC      H'0000FFF0FFF00000'
DC      H'0000000F00000000'
DC      H'0000000F00000000'
DC      H'0000000F00000000'
DC      H'0000000F00000000'
DC      H'0000000F00000000'
DC      H'0000000F00000000'
DC      H'0000000F00000000'
DC      H'0000FFF0FFF00000'
DC      H'0000000000000000'


DC      H'0000FFF0FFF00000'
DC      H'000FFFFFFFFF0000'
DC      H'0000FFFFFFF00000'
DC      H'000000FFF0000000'
DC      H'000000FFF0000000'
DC      H'000000FFF0000000'
DC      H'000000FFF0000000'
DC      H'000000FFF0000000'
DC      H'0000FFFFFFF00000'
```

```
     DC       H'000FFFFFFFFF0000'
     DC       H'0000FFF0FFF00000'


     DC       I2'9,7'                ;Hot spot (base of I-Beam)
```

You can locate the cursor record for the currently active cursor with GetCursorAdr:

```
     PHA                          ;Space for result
     PHA
     _GetCursorAdr
     PopLong CursorPtr            ;Pop the result
```

If you wish to switch cursors temporarily, use GetCursorAdr to get the pointer to the current cursor and then use the pointer when restoring the cursor with SetCursor.

If you want to remove the cursor from the screen for any reason, call HideCursor. To make it visible again, call ShowCursor. Neither function requires parameters or returns results.

If you are using a cursor to identify a position where text may be entered, you may want to make the cursor blink by hiding and showing it at a fixed rate. The blink rate should be the one set by the Control Panel's Cursor Flash command; it can be read using the GetCaretTime function:

```
     PHA                          ;space for result
     PHA
     _GetCaretTime
     PopLong CaretTime
```

The long result is expressed in ticks (sixtieths of a second). The default Control Panel setting is 30 (half a second).

One handy cursor function is ObscureCursor. It removes the cursor from the screen until the mouse moves. Use it to eliminate the mouse cursor while a user is typing something in from the keyboard. It requires no parameters.

The QuickDraw Auxiliary tool set contains a function called WaitCursor that you can use to change the cursor to a wristwatch. You should call it just before performing a time-consuming operation to inform the user that the program will not be responding to input for a while. To restore the arrow cursor, call InitCursor. WaitCursor requires no input parameters and returns no results.

To start up the QuickDraw Auxiliary tool set, call the QDAuxStartup function. To shut it down, call QDAuxShutDown. Neither function requires parameters nor returns results.

CLOCK FUNCTIONS

As mentioned in chapter 1, the GS has a built-in clock/calendar chip that maintains the current time and date. You can set the current time and date with the Control Panel.

One common I/O operation (which, like mouse movement, is not handled by the Event Manager) is reading the time and date so that you can display it on the screen or insert it in a printed report. The Miscellaneous Tool Set includes two functions you can use to read the clock: ReadAsciiTime and ReadTimeHex.

(The start up and shut down functions for the Miscellaneous Tool Set are MTStartup and MTShutDown. Neither one requires parameters nor returns results.)

ReadAsciiTime, as its name suggests, returns a 20-byte ASCII-encoded character string describing the current date and time. To call it, pass a pointer to a previously-allocated 20-byte data area:

```
    PushPtr TimeString                ;Pointer to string area
    _ReadAsciiTime                    ;Read the time
    RTS

    TimeString DS   20                ;Space for time string
```

The time string is not preceded by a length byte. It is always exactly 20 characters long and the high-order bit in each byte is set to one.

The string returned by ReadAsciiTime uses one of three date formats and one of two time formats. Altogether, there are six formatting combinations (dd = day of month, MM = month, yy = year, hh = hours, mm = minutes, and ss = seconds):

```
    dd/MM/yy hh:mm:ss XM        (XM = AM or PM)
    MM/dd/yy hh:mm:ss XM        (this is the default)
    yy/MM/dd hh:mm:ss XM
    dd/MM/yy hh:mm:ss           (24-hour military format)
    MM/dd/yy hh:mm:ss
    yy/MM/dd hh:mm:ss
```

The date and time formats actually used are those set by the Clock command in the Control Panel.

If you are not satisfied with the look of the string that ReadAsciiTime returns, use ReadTimeHex to return the current time and date in binary form and then manipulate the data as you like. Here is how to call ReadTimeHex:

```
    PHA                       ;Space for DayOfWeek/[unused] bytes
    PHA                       ;Space for Month/DayOfMonth
    PHA                       ;Space for Year/Hour
    PHA                       ;Space for Minute/Second
    _ReadTimeHex
    SEP     #$20                ;Switch to 8-bit A for byte data
    LONGA OFF

    PLA                         ;Pop Second
    STA     Second
    PLA                         ;Pop Minute
    STA     Minute
    PLA                         ;Pop Hour
    STA     Hour
    PLA                         ;Pop Year
    STA     Year
    PLA                         ;Pop Day of Month
    STA     DayOfMonth
    PLA                         ;Pop Month
    STA     Month
    PLA                         ;Pop unused byte
    PLA                         ;Pop Day of Week
    STA     DayOfWeek

    REP     #$20                ;Return to 16-bit accumulator
    LONGA   ON
```

The values returned by ReadTimeHex are as follows:

FUNCTION	VALUE
DayOfWeek	1..7 (1=Sunday, 2=Monday, etc.)
Year	0..99 (Year minus 1900)
Month	0..11 (1=January, 2=February, etc.)
DayOfMonth	0..30 (Day of month minus 1)
Hours	0..23
Minutes	0..59
Seconds	0..59

Remember that ReadTimeHex was the function used by the user-defined TimeTools tool set in chapter 3 to determine the day of the week code prior to converting it to a text string.

One other useful time-related function is TickCount. Use it to determine the number of ticks that have elapsed since the GS was started up (a tick is 1/60th of a second) . Here is how to call TickCount:

```
    PHA                         ;Space for long result
    PHA
    _TickCount
    PopLong NumTicks            ;Pop the result
```

To put a delay loop in a program, call TickCount once before entering the loop and then keep calling it until the difference between the result and the starting value is greater than or equal to the desired delay.

To generate a two-second (120-tick) delay, for example, use the following code fragment:

```
          PHA
          PHA
          _TickCount           ;Get starting tick count
          PopLong TickStart


DelayLoop PHA
          PHA
          _TickCount           ;Get current tick count
          PopLong TickCurr


          SEC                  ;Calculate time difference
          LDA   TickCurr
          SBC   TickStart
          STA   TickCurr
          LDA   TickCurr+2
          SBC   TickStart+2
          STA   TickCurr+2


          SEC                  ;Subtract 120 (long)
          LDA   TickCurr
          SBC   #120
          LDA   TickCurr+2
          SBC   #0


          BCC DelayLoop        ;Branch if < 120


          RTS


TickStart DS    4
TickCurr  DS    4
```

Notice that the carry flag, which is set before a subtraction operation, becomes clear if the number being subtracted is greater than or equal to the number in the accumulator.

Table R5–1: The Major Functions in the Event Manager Tool Set ($06)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
Button	$0D	result (W)	Boolean: is the button down?
		ButtonNum (W)	Button number
EMStartup	$02	DPAddr (W)	Address of 1 page in bank 0
		QueueSize (W)	Size of event queue
		XMinClamp (W)	Mouse minimum (horizontal)
		XMaxClamp (W)	Mouse maximum (horizontal)
		YMinClamp (W)	Mouse minimum (vertical)
		YMaxClamp (W)	Mouse maximum (vertical)
		UserID (W)	ID tag for memory allocation
EMShutDown	$03	[no parameters]	
EventAvail	$0B	result (W)	Boolean: always true
		EventMask (W)	Event mask
		EventRecord (L)	Ptr to event record
FlushEvents	$15	result (W)	Mask for next event in queue
		EventMask (W)	Mask for events to be flushed
		StopMask (W)	Masks for events to be kept
GetCaretTime	$12	result (L)	Ticks between cursor blinks
GetDblTime	$11	result (L)	Interval for double-clicks
GetMouse	$0C	MouseLocPtr (L)	Ptr to mouse point (local)
GetNextEvent	$0A	result (W)	Boolean: was event retrieved?
		EventMask (W)	Event mask
		EventRecord (L)	Ptr to space for event record

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
PostEvent	$14	result (W)	Boolean: was event posted?
		EventCode (W)	Event code to place in queue
		EventMsg (L)	Message word for event record
StillDown	$0E	result (W)	Boolean: button still down?
		ButtonNum (W)	Button number
TickCount	$10	result (L)	Ticks since system startup
WaitMouseUp	$0F	result (W)	Boolean: button still down?
		ButtonNum (W)	Button number

Table R5–2: Event Manager Error Codes

Error Code	*Description of Error Condition*
$0601	The Event Manager has already been started up.
$0602	Reset error.
$0603	The Event Manager is not active.
$0604	The event code is greater than 15.
$0605	The specified button number is not 0 or 1.
$0606	The size of the event queue is greater than 3639.
$0607	There is not enough memory for the event queue.
$0681	The event queue is seriously damaged.
$0682	The handle to the event queue is damaged.

Table R5–3: Useful Functions in the Window Manager Tool Set ($0E)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
TaskMaster	$1D	result (W)	Event code
		EventMask (W)	Event mask
		TaskRecord (L)	Ptr to space for task record

Table R5–4: Useful Functions in the QuickDraw II Tool Set ($04)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
GetCursorAdr	$8F	result (L)	ptr to cursor record
HideCursor	$90	[no parameters]	
InitCursor	$CA	[no parameters]	
ObscureCursor	$92	[no parameters]	
SetCursor	$8E	CursorRecord (L)	ptr to cursor record
ShowCursor	$91	[no parameters]	

Table R5–5: Useful Functions in the QuickDraw Auxiliary Tool Set ($12)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
QDAuxShutDown	$03	[no parameters]	
QDAuxStartup	$02	[no parameters]	
WaitCursor	$0A	[no parameters]	

Table R5–6: Useful Functions in the Miscellaneous Tool Set ($03)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
MTShutDown	$03	[no parameters]	
MTStartup	$02	[no parameters]	

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
ReadAsciiTime	$0F	TimeString (L)	Ptr to 20-byte time string
ReadTimeHex	$0D	result (W)	Day of week (high byte)
		result (W)	Month (high), Date (low)
		result (W)	Year-1900 (high), Hour (low)
		result (W)	Minute (high), Second (low)

Listing 5–1: A Program for Assisting in the Development of Other Programs

```
******************************************
* You can use this program as a shell    *
* for writing a complete GS application. *
******************************************
           LIST     OFF
           ABSADDR  ON
           INSTIME  ON
           GEN      OFF
           SYMBOL   OFF

           KEEP     SHELL               ;Object code file
           MCOPY    SHELL.MAC           ;Macro file

MyCode     START
           Using    GlobalData
           Using    StartData

           JSR      DoStartUp           ;Start up all tool sets

; Define and display the menu bar:

           PushLong #0
           PushPtr  MenuL2
           _NewMenu
           PushWord #0
           _InsertMenu

           PushLong #0
           PushPtr  MenuL1
           _NewMenu
           PushWord #0
           _InsertMenu

           PushWord #1
           _FixAppleMenu                ;Add desk accessories to menu

           PHA
           _FixMenuBar                  ;Adjust size of menus/menu bar
           PLA

           _DrawMenuBar

; Define and display the window:

           PHA                          ;Space for result
           PHA
           PushPtr  MainWindow          ;Pointer to window record
           _NewWindow
```

```
            PopLong WindowPtr           ;Save pointer to window record
            _InitCursor                 ;Turn on the arrow cursor

EvtLoop     PHA
            PushWord #$FFFF             ;Allow all events
            PushPtr  EventRec
            _TaskMaster

            PLA                         ;Get result code
            CMP      #wInMenuBar        ;Menu item selected?
            BEQ      DoMenu             ;Yes, so branch

; [check for other events here]

            BRL      EvtLoop            ;Back for more

; Handle menu item selections. (The items must be
; numbered sequentially from 256.)

DoMenu      LDA      TaskData           ;Get menu item ID
            SEC
            SBC      #256               ;Convert to 0 base
            ASL      A                  ;x2 to step into table
            TAX
            JSR      (MenuTable,X)

            PushWord #0                 ;Highlighting off
            PushWord TaskData+2         ;Get menu ID
            _HiliteMenu

            BRA      EvtLoop

; Table of menu item subroutine addresses:

MenuTable   DC       I'DoAboutI'        ;Item 256
            DC       I'DoQuitI'         ;Item 257
;           DC       I'DoYourItem'      ;Item 258 ... and so on

DoAboutI    RTS

DoQuitI     PLA                         ;Pop return address
            JMP      DoShutDown         ;Shut down all tool sets

            END

            COPY     STANDARD.ASM
```

```
GlobalData DATA

; Menu/item line lists. If you add more items, number them
; consecutively from 258.

MenuL1     DC      C'>>@\N1X';H'0D'      ;Apple menu
           DC      C'##About this program ...\N256V',H'0D'

MenuL2     DC      C'>> File \N2',H'0D'  ;File menu
           DC      C'##Quit\N257*Qq',H'0D'
           DC      C'.'                 ;End of menu

; Window parameters:

WindowPtr  DS      4                    ;Pointer to window

MyTitle    STR     ' GS Application '

MainWindow DC      I2'WindEnd-MainWindow'
           DC      I'%1000000010100101'          ;window frame definition
           DC      I4'MyTitle'                   ;Pointer to window title
           dc      I4'0'                         ;refcon
           DC      I'0,0,0,0'                    ;zoom rectangle
           DC      I4'0'                         ;color table
           DC      I'0,0'                        ;origin offset
           DC      I'0,0'                        ;height,width data area
           DC      I'0,0'                        ;height,width max window
           DC      I'0,0'                        ;vert, horiz scroll
           DC      I'0,0'                        ;vert, horiz page
           DC      I4'0'                         ;info bar refcon
           DC      I4'0'                         ;frame defproc
           DC      I4'0'                         ;info bar defproc
           DC      I2'0'                         ;info bar height
           DC      I4'0'                         ;content defproc
WindRect   DC      I'30,10,185,400'              ;Content region rectangle
           DC      I4'-1'                        ;At the front
           DC      I4'0'                         ;Storage
WindEnd    ANOP

EventRec   ANOP
What       DS      2                    ;Event code
Message    DS      4                    ;Event result
When       DS      4                    ;Ticks since startup
Where      DS      4                    ;Mouse location (global)
Modifiers  DS      2                    ;Status of modifier keys
TaskData   DS      4                    ;TaskMaster data
TaskMask   DC      I4'$00001FFF'

           END
```

Listing 5–2: A Program Showing How to Use StillDown and GetMouse

```
*******************************************
* This program shows how to use the       *
* StillDown and GetMouse functions. By    *
* pulling on the mouse with the button    *
* down, you can create a rectangle of     *
* any size.                               *
*******************************************
           LIST    OFF
           ABSADDR ON
           INSTIME ON
           GEN     OFF
           SYMBOL  OFF

           KEEP    DRAG                ;Object code file
           MCOPY   DRAG.MAC            ;Macro file

MyCode     START
           Using   GlobalData

           JSR     DoStartUp

; Define and display the window:

           PushLong #0                 ;Space for result
           PushPtr  MainWindow         ;Pointer to window record
           _NewWindow
           _StartDrawing               ;Prepare to draw in window

           _InitCursor

EvtLoop    PHA
           PushWord #$FFFF             ;All events
           PushPtr  EventRec
           _TaskMaster
           PLA                         ;Get result code

           CMP     #keyDownEvt         ;Key-down?
           BEQ     Exit                ;Yes, so branch

           CMP     #wInContent         ;in content region?
           BNE     EvtLoop             ;no, so ignore

           JSR     DoBoxes
           BRA     EvtLoop

Exit       JMP     DoShutDown

           END
```

```
; Display a box on the screen. Size it using
; dragging techniques.

DoBoxes    START
           Using   GlobalData

           PushPtr Where
           _GlobalToLocal             ;convert to window coords

; Initial rectangle is a single pixel:

           LDA     Where+h
           STA     BoxRect+left
           STA     BoxRect+right
           LDA     Where+v
           STA     BoxRect+top
           STA     BoxRect+bottom

StretchIt  PushPtr BoxRect
           _InvertRect                ;draw the new rectangle

StretchIt1 PHA                        ;space for result
           PushWord #0                ;mouse button
           _StillDown
           PLA
           BEQ     DragExit           ;branch if not still down

           PushPtr MouseLoc           ;Get current mouse position
           _GetMouse                  ; (local coords)

; If current location is same as last one, do nothing. This
; avoids excessive flashing on the screen.

           LDA     MouseLoc+h
           CMP     LastLoc+h          ;Horizontal matches?
           BNE     SaveLoc            ;No, so branch
           LDA     MouseLoc+v
           CMP     LastLoc+v          ;Vertical matches?
           BEQ     StretchIt1         ;Yes, so branch

SaveLoc    LDA     MouseLoc+h         ;Save the new position
           STA     LastLoc+h
           LDA     MouseLoc+v
           STA     LastLoc+v

           PushPtr BoxRect
           _InvertRect                ;erase the old rectangle

; Arrange coordinates of the new rectangle in the
; standard top, left, bottom, right (TLBR) order:
```

```
            LDA     MouseLoc+h          ;Is new horizontal position
            CMP     Where+h             ; to the left of the base point?
            BCC     _1                  ;Yes, so branch

            STA     BoxRect+right
            LDA     Where+h
            STA     BoxRect+left
            BRA     _2

_1          STA     BoxRect+left
            LDA     Where+h
            STA     BoxRect+right

_2          LDA     MouseLoc+v          ;Is new vertical position
            CMP     Where+v             ; above the base point?
            BCC     _3                  ;Yes, so branch

            STA     BoxRect+bottom
            LDA     Where+v
            STA     BoxRect+top
            BRL     StretchIt

_3          STA     BoxRect+top
            LDA     Where+v
            STA     BoxRect+bottom
            BRL     StretchIt

DragExit    RTS

            END

            COPY    STANDARD.ASM

GlobalData DATA

; Window parameters:

MyTitle     STR     ' Drag Demo '

MainWindow DC       I2'WindEnd-MainWindow'
            DC      I'%1000000000100000'        ;title bar only
            DC      I4'MyTitle'                 ;Pointer to window title
            dc      I4'0'                       ;refcon
            DC      I'0,0,0,0'                  ;zoom rectangle
            DC      I4'0'                       ;color table
            DC      I'0,0'                      ;initial origin offset
            DC      I'0,0'                      ;height,width data area
            DC      I'0,0'                      ;height,width max window
            DC      I'0,0'                      ;vert, horiz line
            DC      I'0,0'                      ;vert, horiz page
            DC      I4'0'                       ;info bar refcon
```

```
           DC      I2'0'                        ;info bar height
           DC      I4'0'                        ;frame defproc
           DC      I4'0'                        ;info bar defproc
           DC      I4'0'                        ;content defproc
WindRect   DC      I'30,10,185,500'             ;Content region rectangle
           DC      I4'-1'                       ;At the front
           DC      I4'0'                        ;Storage
WindEnd    ANOP

MouseLoc   DS      4                    ;a point
BoxRect    DS      8                    ;Coordinates of stretched rectangle
LastLoc    DS      4                    ;a point

EventRec   ANOP
What       DS      2                    ;Event code
Message    DS      4                    ;Event result
When       DS      4                    ;Ticks since startup
Where      DS      4                    ;Mouse location (global)
Modifiers  DS      2                    ;Status of modifier keys
TaskData   DS      4                    ;TaskMaster data
TaskMask   DC      I4'$00001FFF'

           END
```

Listing 5–3: A Program Showing How to Use GetDblTime and WaitMouseUp

```
**************************************************
* This program shows how to detect a double-click *
* operation. When you double-click, it displays a *
* black rectangle on the screen.                  *
**************************************************
           LIST    OFF
           SYMBOL  OFF
           ABSADDR ON
           INSTIME ON
           GEN     ON

           KEEP    DOUBLE               ;Object code file
           MCOPY   DOUBLE.MAC           ;Macro file

MyCode     START
           Using   GlobalData
           Using   StartData

           JSR     DoStartUp

; Define and display the window:
ShowWind   PHA                          ;Space for result
           PHA
           PushPtr  MainWindow          ;Pointer to window record
           _NewWindow
           PopLong WindowPtr       ;Save pointer to window record

           _InitCursor

EvtLoop    PHA
           PushWord #$FFFF         ;All events
           PushPtr  TaskRec
           _TaskMaster
           PLA                     ;Get result code

           CMP     #keyDownEvt     ;Key-down?
           BEQ     Exit            ;Yes, so branch

           CMP     #wInContent     ;in content region?
           BNE     EvtLoop         ;no, so ignore

           JSR     DoClick
           BRA     EvtLoop

Exit       JMP     DoShutDown

           END

; Check for a double-click.
;
; If it's a double click, display a rectangle
; filled with the pen pattern.

DoClick    START
           Using   GlobalData

           PushLong WindowPtr
           _StartDrawing           ;Prepare to draw in window

           PHA                     ;Space for result
           PHA
           _GetDblTime             ;get double-click time
           PopLong  DblTime

           PushPtr Where           ;Convert mouse-down location
           _GlobalToLocal          ; to window coordinates

; Calculate time since last mouse-up:

           SEC
           LDA     When
           SBC     UpTime
           STA     TheGap
           LDA     When+2
           SBC     UpTime+2
           STA     TheGap+2
```

```
; Check the mouse-up / mouse-down interval:

            SEC
            LDA      DblTime
            SBC      TheGap
            LDA      DblTime+2
            SBC      TheGap+2
            BCS      Doubled              ; branch if DblTime >= TheGap
; Wait until mouse button is released, then save time:

WaitForUp   PHA                           ;space for result
            PushWord #0                   ;mouse button (#0)
            _WaitMouseUp                  ;Wait for the following mouse-up event
            PLA                           ;Get Boolean result
            BNE      WaitForUp            ;Loop until mouse is up

            PHA                           ;space for result
            PHA
            _TickCount                    ;Get current tick count
            PopLong  UpTime               ;Get the result
            RTS

; We had a double-click, so draw a solid rectangle:

Doubled     ANOP

            CLC
            LDA      Where+h
            STA      OurBox+left
            ADC      #30
            STA      OurBox+right
            CLC
            LDA      Where+v
            STA      OurBox+top
            ADC      #10
            STA      OurBox+bottom

            PushPtr OurBox
            _PaintRect                    ;Draw the solid box

            RTS

DblTime     DS       4                    ;LONG (double-click time)
UpTime      DC       I4'0'                ;time of last mouse-up
TheGap      DS       4                    ;up-down time gap
OurBox      DS       8                    ;Rectangle

            END
```

```
          COPY     STANDARD.ASM

GlobalData DATA

; Window parameters:

WindowPtr  DS       4                     ;Pointer to window record

MyTitle    STR      ' Double-Click Demo '

MainWindow DC       I2'WindEnd-MainWindow'      ;Size of table
           DC       I'%1101110110100101'        ;window frame type
           DC       I4'MyTitle'                 ;Pointer to window title
           DC       I4'0'                       ;refcon
           DC       I'0,0,0,0'                  ;zoom rectangle
           DC       I4'0'                       ;color table (0 = default)
           DC       I'0,0'                      ;document offset
           DC       I'0,0'                      ;height,width of data area
           DC       I'0,0'                      ;height,width max window
           DC       I'0,0'                      ;vert, horiz line movement
           DC       I'0,0'                      ;vert, horiz page movement
           DC       I4'0'                       ;info bar refcon
           DC       I2'0'                       ;info bar height
           DC       I4'0'                       ;frame defproc (0 = standard
           DC       I4'0'                       ;info bar defproc
           DC       I4'0'                       ;content defproc
           DC       I'31,6,182,608'             ;Content region rectangle
           DC       I4'-1'                      ;At the front
           DC       I4'0'                       ;Storage (use MM)
WindEnd    ANOP

;TaskMaster task record:

TaskRec    ANOP
What       DS       2                     ;Event code
Message    DS       4                     ;Event result
When       DS       4                     ;Ticks since startup
Where      DS       4                     ;Mouse location (global)
Modifiers  DS       2                     ;Status of modifier keys
TaskData   DS       4                     ;TaskMaster data
TaskMask   DC       I4'$00001FFF'         ;TaskMaster handles all

           END
```

CHAPTER 6

Windows and Graphics

This chapter, and the three that follow, examine ways to develop applications that use the GS's super high-resolution graphics screen as a Macintosh-like desktop. The discussion will begin with an analysis of windows and then will move on to pull-down menus, dialog and alert boxes, and finally desk accessories. The GS has tool sets for dealing with all these standard desktop features.

A window is an object on the desktop that acts as the private display screen for a single document. The underlying document may be a series of lines of text or any arbitrary graphic image, and it may be any size. If the underlying document is larger than the portion of the window in which drawing operations appear (called the *content region*), however, not all of it can be seen at once. To permit the viewing of any portion of a large document, you can add bottom and right scroll controls to a window. By adjusting these controls, it is possible to move any portion of the document into the content region.

The GS handles windows in a flexible way. You can define as many windows as memory permits and you can display them all on the screen at once. A window is totally independent of any other window, so it can appear anywhere on the screen, even though it may overlap or totally obscure another window. Some windows are resizable and some may be dragged around the screen with the mouse.

By convention, the frontmost window is the active window, the one that will be affected by drawing operations; other windows are said to be inactive. The active window is drawn in a distinctive way, with a solid block in the title bar and highlighted scroll controls. This distinguishes it from inactive windows, which show only the outlines of the scroll controls and title bar.

This chapter covers how the Window Manager (tool set 14) can be used to create various types of windows and to display them on the screen. Changing the size and appearance of windows, moving windows around the screen, handling windows that have scroll controls, and dealing with multiple windows are also discussed.

Of course, windows are not very useful unless you can display something in them. At the end of the chapter, many of the text and graphics drawing functions in the QuickDraw II tool set will be examined. QuickDraw II is responsible for handling all tasks that involve drawing something on the super high-resolution graphics screen. The Window Manager, for example, uses QuickDraw II to draw window frames and special window icons.

THE SUPER HIGH-RESOLUTION GRAPHICS SCREEN

Before discussing the Window Manager, it is necessary to take a close look at the characteristics of the super high-resolution graphics video display mode. Once you understand how it works, you will be better able to use many of the Window Manager and QuickDraw II functions, especially the ones that permit you to manipulate colors.

As shown in figure 6–1, the video display buffer for the super high-resolution graphics mode is located from $2000 to $9CFF in bank $E1 of memory. The area from $9D00 to $9DFF holds 200 scanline control bytes and the area from $9E00 to $9FFF holds 16 color palettes. The area from $2000 to $9FFF in bank $01 shadows to the same area in bank $E1 if shadowing is enabled. (QuickDraw II does not enable shadowing; it accesses bank $E1 video locations directly.)

Two bits in the New-Video register at $E0C029 control the characteristics of the super high-resolution screen (see figure 6–2).

To turn on the super high-resolution display mode, set bit 7 of the New-Video register to 1. Bit 6 must also be set so there will be a linear relationship between the display buffer and the position of a pixel on the screen. You want the linear relationship because screen drawing calculations can be done more quickly. QuickDraw II sets both bits for you when you call the QDStartup function.

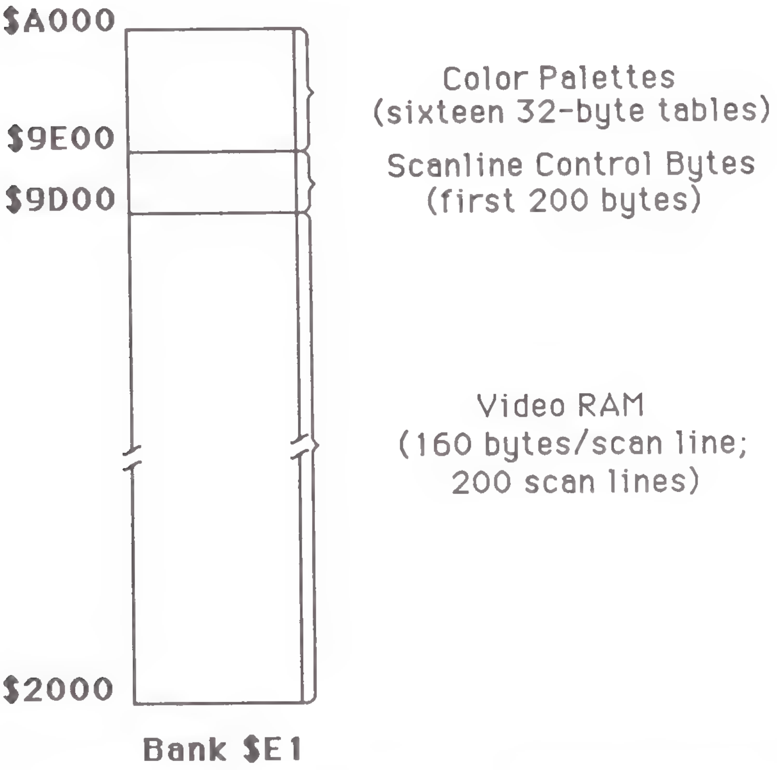
When you leave super high-resolution mode, store a $01 in the New-Video register. This ensures that linearization is off, as normal IIe-style applications expect this. (It also permits main/auxiliary memory switching just like on a IIe.) QuickDraw does this when you call QDShutDown.

Pixel dimensions of the graphics screen are either 320 wide by 200 high or 640 by 200. Each line is defined by a group of 160 consecutive bytes in the video display buffer. With the linear memory map enabled, this means that the bytes defining a line begin at an offset (from $2000) that is 160 times the line number. The line numbers range from 0 to 199.

Each pixel in a line is associated with either four bits (320 mode) or two bits (640 mode) in the display buffer. These bits define the color of the pixel. The assignment of pixels to bits for a byte in the display buffer is shown in figure 6–3.

When this scheme is used, a pixel can be one of sixteen colors in 320 mode or one of four colors in 640 mode. As will be shown below, the color definitions are

Figure 6–1. The Super High-resolution Graphics Display Buffer



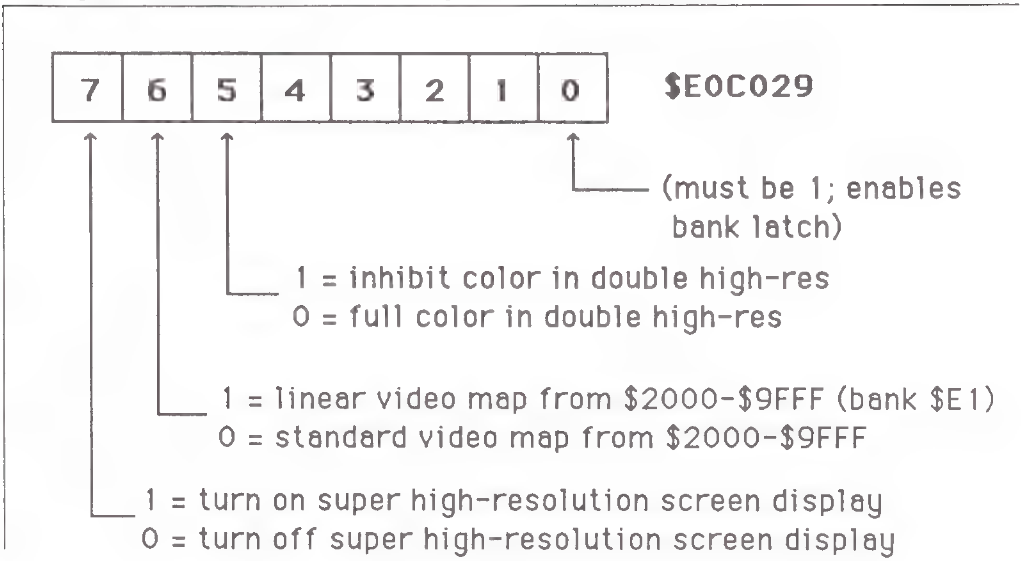
NOTE. Locations $2000—$9FFF in bank $01 shadow to this bank if shadowing is enabled.

stored in a palette, or color table, that defines sixteen colors. In 640 mode, only four of these colors are available to a given pixel; the four that are available depend on what column the pixel is in, as indicated in the figure.

The pixel width and color palette for a given horizontal line are controlled by its *scanline control byte* (SCB). The SCBs for the screen are stored, in line order, in a 200-byte table beginning at $9D00 in bank $E1. The format of an SCB is shown in figure 6–4.

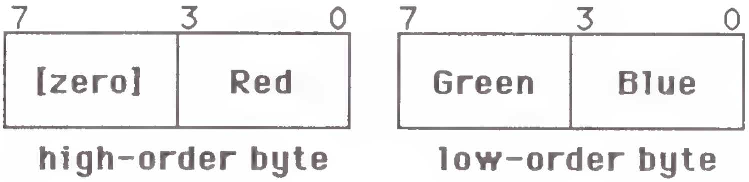
When the fill mode bit of an SCB is on, color number 0 in the palette becomes inactive. When a pixel is assigned a color number of 0, the pixel takes on the same color as the previous non-zero pixel in the line.

Figure 6–2. The New-video Register


7
6
5
4
3
2
1
0
$E0C029
(must be 1; enables bank latch)
1 = inhibit color in double high-res
0 = full color in double high-res
1 = linear video map from $2000-$9FFF (bank $E1)
0 = standard video map from $2000-$9FFF
1 = turn on super high-resolution screen display
0 = turn off super high-resolution screen display

The interrupt bit in the SCB is normally off. You may want to enable interrupts so that you can update the video buffer during video retrace operations; this eliminates flickering in animation sequences. You could also change the color palettes during an interrupt to effectively increase the number of different colors visible on the screen.

A palette, or color table, is a group of sixteen words, each describing a particular color, as follows:



As you can see, the red, green, and blue components of the color word are each 4 bits long. This means you can choose from 4,096 different colors.

Table 6–1 shows the standard color tables QuickDraw II uses for each of the sixteen 16-color palettes when it starts up in the 640-by-200 and the 320-by-200 mode.

Figure 6–3. Super High-resolution Color Bits

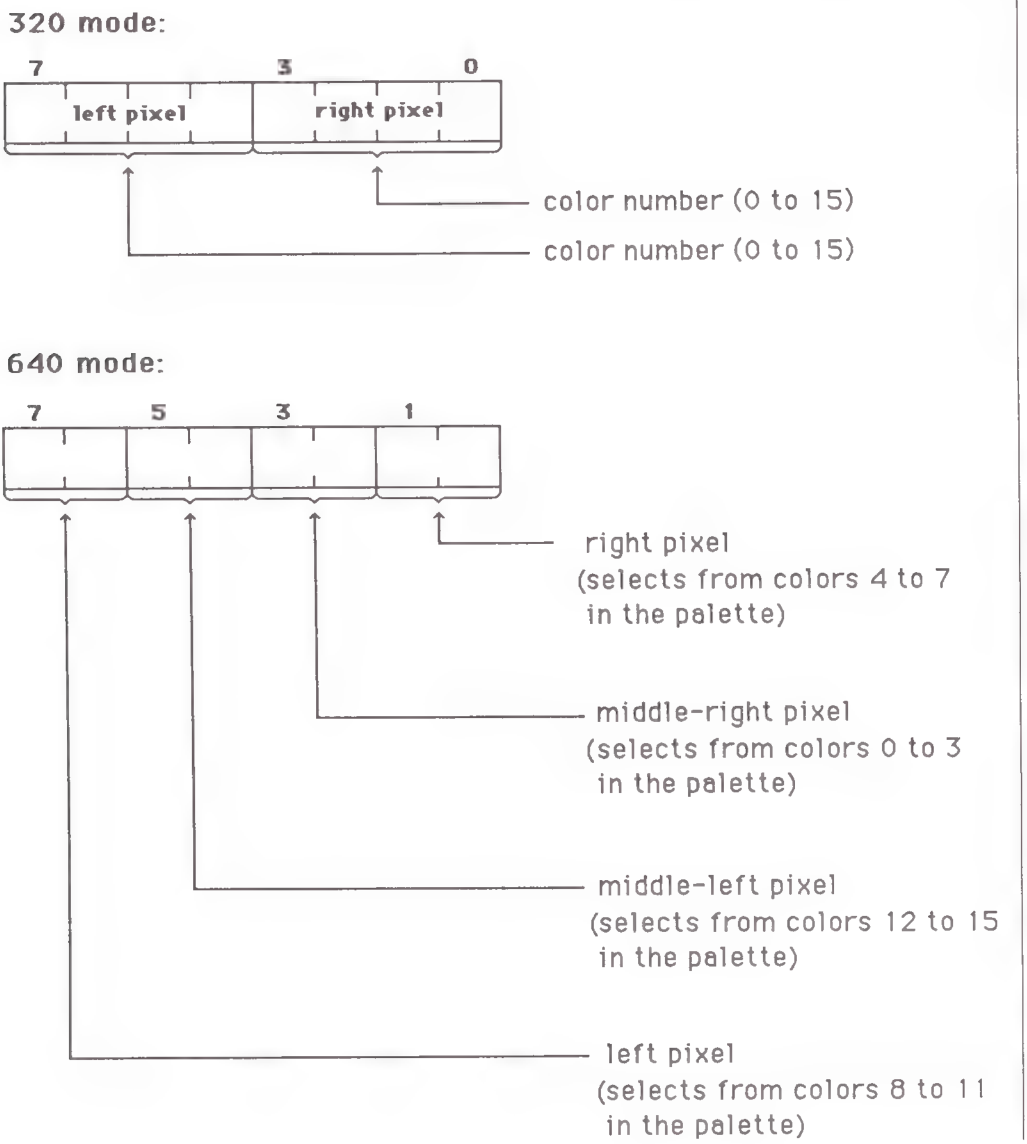
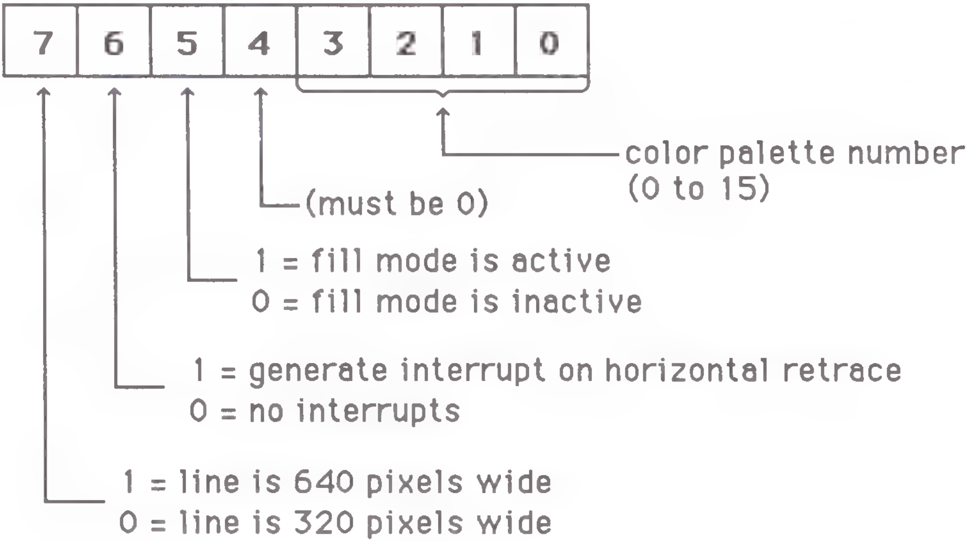


Figure 6–4. The Format of a Scanline Control Byte


7
6
5
4
3
2
1
0
color palette number (0 to 15)
(must be 0)
1 = fill mode is active
0 = fill mode is inactive
1 = generate interrupt on horizontal retrace
0 = no interrupts
1 = line is 640 pixels wide
0 = line is 320 pixels wide

You can return a copy of this table by pushing a pointer to a sixteen-word area and calling the QuickDraw InitColorTable function.

In 640 mode, QuickDraw expects color numbers from 0 to 3 only. In this situation, a color number is really an index into the group of four colors permitted for the pixel column in question. Notice that the four groups of four colors in 640 mode are different—two of the groups use red and green as the second and third colors and two use blue and yellow. This means that when a horizontal line in color $02 is drawn, alternate green and yellow pixels are drawn on the screen; they are so close together, however, that the line appears to be one solid color (lime green). An alternate red/green line looks purple. The creation of a new color by placing different colors close to one another is called *dithering*. By changing the 640-mode palette and taking advantage of the dithering phenomenon, you can create many other colors.

Any entry in any palette table can be changed at any time using QuickDraw's SetColorEntry function:

```
PushWord #0              ;Palette number (0 to 15)
PushWord #8              ;Color number (0 to 15)
PushWord NewColor        ;The new color value
_SetColorEntry
```

Table 6–1: The Standard Color Tables Used by QuickDraw

Color	*320-by-200 Mode*		*640-by-200 Mode*	
Number	*Value*	*Color*	*Value*	*Color*
0	$0000	black	$0000	black
1	$0777	dark gray	$0F00	red
2	$0841	brown	$00F0	green
3	$072C	purple	$0FFF	white
4	$000F	blue	$0000	black
5	$0080	dark green	$000F	blue
6	$0F70	orange	$0FF0	yellow
7	$0D00	red	$0FFF	white
8	$0FA9	flesh	$0000	black
9	$0FF0	yellow	$0F00	red
10	$00E0	green	$00F0	green
11	$04DF	light blue	$0FFF	white
12	$0DAF	lilac	$0000	black
13	$078F	periwinkle blue	$000F	blue
14	$0CCC	light gray	$0FF0	yellow
15	$0FFF	white	$0FFF	white

If you prefer, you can redefine the entire table at once by passing a palette number and a pointer to the palette definition to SetColorTable.

There are also QuickDraw functions for changing the standard scanline control bytes: SetSCB and SetAllSCBs. Use SetSCB to change the SCB for any particular line, or SetAllSCBs to assign the same SCB to all lines at once.

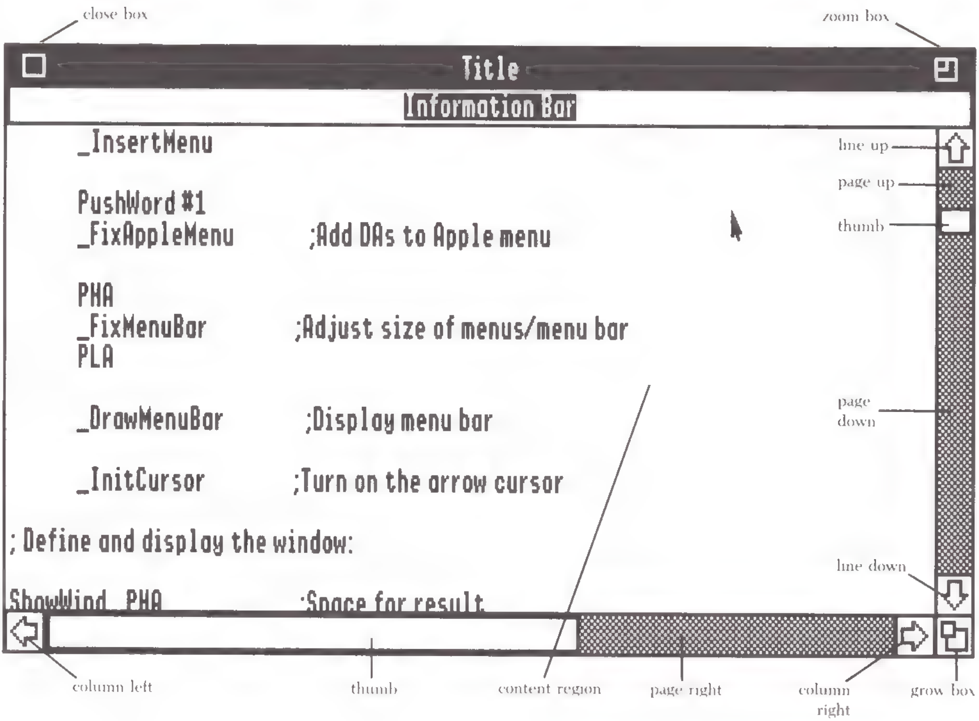
Here is how to use SetSCB:

```
PushWord #188            ;Line number
PushWord #$82            ;New SCB
_SetSCB
```

In this example, line #188 becomes 640 pixels wide and is associated with palette #2.

Because the SetAllSCBs function affects all lines, it requires only one parameter, the new SCB value.

Figure 6–5. The Structure of a Window



```
        _InsertMenu

        PushWord #1
        _FixAppleMenu       ;Add DAs to Apple menu

        PHA
        _FixMenuBar         ;Adjust size of menus/menu bar
        PLA

        _DrawMenuBar        ;Display menu bar

        _InitCursor         ;Turn on the arrow cursor

; Define and display the window:

ShowWind  PHA               ;Space for result
```

INTRODUCING THE WINDOW

As shown in figure 6–5, a standard window is made up of a rectangular *content region* in which the window's underlying document is drawn. Surrounding the content region is a *window frame* in which any *window controls* defined for the window appear. Methods for assigning window controls to a window are covered later in this chapter. The window in figure 6-5 is "fully-loaded"; its frame contains every type of control supported by the Window Manager. In practice, most windows use only a subset of these controls.

Just above the content region of this window is the *information bar*. Most windows do not have an information bar; if one is defined, it usually contains a subtitle for the window. The main title appears in a *title bar* above the information bar. The title bar also serves as the mechanism for dragging a window around the screen. To drag a window, move the mouse cursor into the title bar, press the mouse button and then move the mouse while holding the button down; to "drop" the window into place, release the mouse button.

Two small rectangles, the *close box* and *zoom box*, appear at the left and right sides of the title bar, respectively. (You must include a title bar if you want a close box or a zoom box.) Although these boxes are inside the rectangle describing the title bar, they are not actually part of the title bar itself. The close box is what a user clicks to remove a window from the screen. The zoom box lets a user quickly resize a window: when it is clicked for the first time, the window expands to cover the entire screen (or whatever size is specified by the programmer); when it is clicked again, it returns to its previous size.

A user can also resize a window by dragging in the *grow box* region of a window. When it is dragged, an outline of the resized window (with the top-left corner fixed in place) follows the mouse until the button is released.

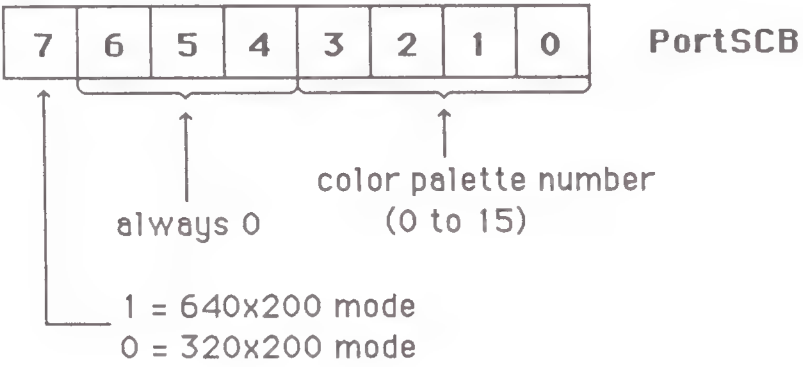
The *scroll bars* permit a user to view any portion of a window's underlying document. When you click the mouse in a *scroll arrow* or a *page region*, or when you drag a *thumb*, whatever is in the window scrolls and a new portion of the document becomes visible. Consider how the right (vertical) scroll bar works, for example. When you click the up or down arrow, the document scrolls up or down by a fixed number of pixel lines corresponding to one line of whatever is in the document. For a text document, this number is usually set to the line spacing. Clicks in the page regions usually scroll the document by the entire height of the content region, or the entire height less one line. Finally, you can quickly bring any portion of the document into view by dragging the scroll bar's thumb up or down in the scroll bar's elevator shaft. Bottom (horizontal) scroll bars behave in similar ways, except that they control horizontal movement within the document.

The size of the thumb reflects how much of the document is being shown in the content region. Again, consider a right scroll bar. The ratio of the height of the thumb to the height of the scroll bar is the same as the ratio of the height of the content region to the height of the document. If the entire height of the document fits in the content region, the thumb occupies the entire scroll bar.

The position of the thumb within the scroll bar tells you what portion of the document is in the content region. The closer the thumb is to the top of a right scroll bar (or to the left of a bottom scroll bar), for example, the closer you are to the top edge (or left edge) of the document. In effect, the ratio of the size of a page region in a scroll bar to the size of the bar is the same as the ratio of the size of the relevant off-screen portion of a window (the portion above the top, bottom, left, or right, as the case may be) to the size of the entire window.

The window behavior just described does not happen automatically. It is up to the programmer to write the application in such a way that standard actions occur when window activity is detected.

Figure 6–6. The Format of the PortSCB Byte



THE WINDOW RECORD

When you first define a window (with the NewWindow function), a window record is created that the Window Manager uses to keep track of window parameters. One important field in a window record is a QuickDraw data structure called a *GrafPort*. A GrafPort completely describes a drawing environment: the position of the drawing area, the attributes of the drawing pen, the character font to be used for drawing text, the typeface and color, the background pattern, and so on.

The first part of a GrafPort is called the PortInfo field. It describes the position and size of a rectangular area in memory within which an active drawing area is defined. In most cases, this area will be inside the super high-resolution video buffer so that you can see the effect of drawing operations.

PortInfo is a data structure of type LocInfo and is made up of the following five fields:

- PortSCB (byte)
- [Reserved] (byte)
- PointerToPixelImage (long)
- Width (word)
- BoundsRect (rectangle)

PortSCB defines the color palette associated with the GrafPort and the bit size of the pixels in the drawing area. Its meaning is explained in figure 6–6.

Although you can associate one of sixteen color palettes with each line in the drawing area (which is usually the super high-resolution display buffer), QuickDraw initially assigns the same palette (palette #0) to each line. (The structure of a palette was described earlier in this chapter.)

PointerToPixelImage contains the address of the top left-hand corner of a rectangular desktop in which the active drawing area of the GrafPort is located. For the super high-resolution screen, this is location $E120000. Width is the number of bytes required to define a line of pixels extending from the left edge of the drawing area to the right edge; it is 160 for the super high-resolution screen.

The precise width of the drawing area is defined by the boundary rectangle, BoundsRect. Like any QuickDraw rectangle, this one is defined by four integers representing the positions of the top, left, bottom, and right sides of the rectangle.

Another important field in a window record is PortRect, which is also a rectangle. PortRect describes the portion of BoundsRect to which GrafPort drawing operations will be confined. If part of PortRect lies outside BoundsRect, drawing will be restricted to the portion inside BoundsRect. For a window, this is the content region.

To determine the current value of PortRect, use GetPortRect:

```
          PushPtr WindRect    ;Pointer to rectangle
          _GetPortRect        ;Read the value of PortRect
          RTS

  WindRect DS  8              ;A rectangle is 4 words (8 bytes)
```

PortRect also defines a *local coordinate system* for the GrafPort, a system used by all drawing functions affecting the GrafPort. In this system, the origin of the GrafPort—its top left-hand corner—has the coordinates given by the first two words of the PortRect. These coordinates are anchored to this corner and do not change even if PortRect is moved from place to place within the area of memory described by the boundary rectangle (i.e., the desktop).

If you do want to change the coordinates of the origin, perhaps to the convenient (0,0) standard or to the scroll position within the document, use the SetOrigin function:

```
  PushWord NewX           ;new horizontal coordinate
  PushWord NewY           ;new vertical coordinate
  _SetOrigin
```

SetOrigin assigns the specified values to the top left corner of PortRect. The bottom right coordinate of PortRect is adjusted accordingly to maintain the size of PortRect. BoundsRect is also adjusted by SetOrigin to keep it at the same relative position to PortRect.

The other standard coordinate system, the *global coordinate system*, assigns the top left-hand corner of the total drawing area given by BoundsRect (usually the super high-resolution screen) to the (0,0) coordinate. This is the coordinate system used by a point in the Where field of an event record, for example. Use the GlobalToLocal function to convert a global coordinate to a local coordinate:

```
    PushPtr ThePoint          ;Pointer to global coordinate
    _GlobalToLocal
```

To convert in the opposite direction, from local to global, use LocalToGlobal:

```
    PushPtr ThePoint          ;Pointer to local coordinate
    _LocalToGlobal
```

Use LocalToGlobal when you have to compare points expressed in the local coordinates of different GrafPorts. Converting all the points to the same coordinate system will enable you to make meaningful comparisons.

The conceptual drawing space for QuickDraw operations is a square grid that is 32K pixels wide and 32K pixels high. The coordinate origin is in the center, so the coordinate limits for the horizontal and vertical axes range from -16K to +16K. If you try to draw outside this range, the system will behave unpredictably.

Keep in mind that even when you draw inside the absolute boundaries only the bits for pixels falling within the content region of the window (and not obscured by other windows) will be transferred to the memory area associated with the content region.

WINDOW MANAGER START-UP AND SHUT-DOWN OPERATIONS

Because the Window Manager functions use QuickDraw II functions to perform screen-drawing operations, you must initialize QuickDraw before initializing the Window Manager. To do this, use the following calling sequence:

```
    PushWord DPAddr           ;Address of 3-page area in bank $00
    PushWord #$80             ;$80 = 640x200 / $00 = 320x200
    PushWord #0               ;MaxWidth (0 = screen width)
    PushWord UserID           ;ID returned by MMStartup
    _QDStartup
```

DPAddr is the two-byte address of a three-page area in bank $00 that QuickDraw uses for direct page space. This space can be allocated with the Memory Manager. The second parameter pushed, MasterSCB, is the scanline control byte QuickDraw will use for each of the 200 lines on the screen. It is normally $80 for 640-by-200 mode or $00 for 320-by-200 mode. The next word pushed is MaxWidth, the width (in bytes) of the largest pixel area in which QuickDraw will be asked to draw; a value of 0 sets MaxWidth to the screen width (160 bytes). The last parameter is the program ID that was returned by MMStartup.

To initialize the Window Manager itself, use WindStartup:

```
    PushWord UserID              ;ID returned by MMStartup
    _WindStartup
```

WindStartup's only parameter, UserID, is the ID number returned by MMStartup.

WindStartup clears the Window Manager's internal list of windows and defines the entire super high-resolution screen as the Window Manager's desktop. It does not draw the desktop itself; for that use RefreshDesktop:

```
    PushLong #0                  ;0 = entire screen
    _RefreshDesktop
```

RefreshDesktop uses a default background pattern and color to draw the desktop.

When you are all through with the Window Manager at the end of a program, call the WindShutDown function to release the data structures and memory areas it uses. WindShutDown requires no parameters.

CREATING A WINDOW

To create a new window, use NewWindow. It builds a window record data structure from a specified parameter list and adds the record to an internal list of window records maintained by the Window Manager. It may or may not display the window on the screen, depending on how you initially define the WFrame parameter (see below). If you define it as invisible, you can display it by passing the window pointer to the ShowWindow function.

NewWindow requires only one parameter, a pointer to a parameter list describing the properties of the window. It returns a pointer to the window record (or 0 if space for the window record could not be allocated). Here is how to use NewWindow:

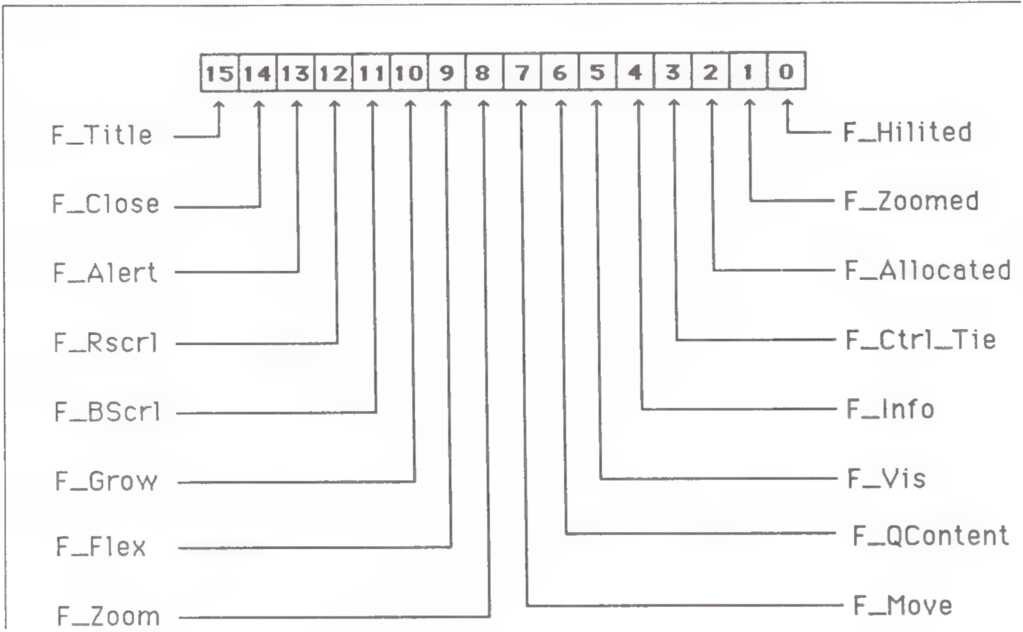
```
    PHA                          ;Space for result (long)
    PHA
    PushPtr WindParms            ;Pointer to parameter list
    _NewWindow
    PopLong WindowPtr            ;Save window pointer
```

NewWindow Parameters

The parameter list referred to in the call to NewWindow contains nineteen parameters that describe the appearance of the window. Some parameters are only for the use of the TaskMaster function, which retrieves events and handles mouse-down activity in a window. The subroutine in listing 6–1 shows how to define the window shown in figure 6–5.

A description of each of the parameters, in order of appearance in the parameter list, follows:

Figure 6–7. The Window Frame Bit Vector (wFrame) Used by NewWindow



paramlength **(word).** This is the number of bytes in the window parameter table.

wFrame **(word).** This is a bit vector describing the appearance of the window frame (see figure 6–7). It lets you create windows with or without the following window controls in the frame:

- Title bar
- Close box
- Zoom box
- Right scroll bar
- Bottom scroll bar
- Grow box
- Information bar

You can also use it to indicate whether the window is movable (can be dragged around the screen), at its zoomed size, or visible. Additional bits reflect whether the window was created by NewWindow, whether the controls are tied to the window, and whether a mouse click that activates a window is to be returned to the application.

With two exceptions, each control is independent of any other. The first exception is the grow box—if its bit is set, the right scroll bar or bottom scroll bar bit must also be set. The second exception relates to the title bar—if the close box or zoom box bits are set, the title bar bit must be set as well. Although it is not required, if you define a window with a zoom box, you should also define a grow box so that the window can be made any convenient size.

A common window frame value is %1101111111100101 ($DFE5). This defines a visible window that has a title bar, close box, scroll bars, grow box, and a zoom box. The window is also draggable.

Here is the meaning of each bit in the window frame bit vector:

bit 0
F_HILITED
1 = Window is highlighted (active)
0 = Window is unhighlighted (inactive)
This bit is used by the Window Manager after the window is defined; its value is ignored by NewWindow.

bit 1
F_ZOOMED
1 = Window is zoomed out
0 = Window is zoomed in
If the window is zoomed out, it will zoom in when the zoom box is clicked next. If it is zoomed in, it will zoom out.

bit 2
F_ALLOCATED
1 = Window record was allocated by NewWindow
0 = Window record was allocated by the application
If this bit is set (the usual case), the window record is automatically disposed of when CloseWindow is called.

bit 3
F_CTRL_TIE
1 = Controls in the window are not tied to the active/inactive state of the window
0 = Controls are tied to the state of the window
The usual value for this bit is 0. This causes the scroll controls to be dimmed when the window is deactivated.

bit 4
F_INFO
1 = The window has an information bar
0 = The window does not have an information bar
If this bit is set, the wInfoDefProc field of the window record must point to an information bar drawing procedure. The field must not be zero.

bit 5
F_VIS
1 = The window is visible
0 = The window is invisible
If this bit is 0, the window will not appear on the screen after NewWindow is called. Use ShowWindow to make the window visible.

bit 6
F_QCONTENT

1 = TaskMaster handles a mouse-down event in an inactive window by activating the window and retaining the event for the application
0 = TaskMaster returns a null event to the application after handling a mouse-down event to activate a window

If this bit is set to 1, the user does not have to click the mouse twice to select something in an inactive window.

bit 7
F_MOVE

1 = The window can be dragged by its title bar
0 = The window cannot be dragged

If a window has a title bar, this bit should be set to 1 so that the user is free to position it on the screen wherever he likes.

bit 8
F_ZOOM

1 = The window has a zoom box
0 = The window does not have a zoom box

If this bit is set, bit 15 (F_TITLE) must also be set so that the window will have a title bar.

bit 9
F_FLEX

1 = The origin of the window does not change when the window is zoom or resized
0 = The origin of the window moves up and to the right as the window grows beyond the limits of its underlying data

For most applications, this bit should be set to 1.

bit 10
F_GROW

1 = The window has a grow box
0 = The window does not have a grow box

A window with a grow box should also have a right or a bottom scroll bar. A zoom box is also recommended but not necessary.

bit 11
F_BSCRL

1 = The window has a bottom scroll bar
0 = The window does not have a bottom scroll bar

bit 12
F_RSCRL

1 = The window has a right scroll bar
0 = The window does not have a right scroll bar

bit 13
F_ALERT

1 = The window uses an alert box window frame
0 = The window uses a standard window frame

You will usually clear this bit to 0 for windows defined with NewWindow. Alert boxes are created using the Dialog Manager. If the bit is 1, F_GROW, F_CLOSE, F_INFO, F_TITLE, F_BSCRL, and F_RSCRL must all be 0.

bit 14
F_CLOSE

1 = The window has a close box
0 = The window does not have a close box

If this bit is set, bit 15 (F_TITLE) must also be set so that the window will have a title bar.

bit 15 F_TITLE	1 = The window has a title bar 0 = The window does not have a title bar

***wTitle* (long).** This is a pointer to the title of the window. The title is a string of ASCII-encoded characters preceded by a length byte. To isolate the title from the background pattern of the title bar include a space character at the beginning and end of the title string.

***wRefCon* (long).** This is a reference constant defined by the application. It is not used by any Window Manager functions. It can hold anything you like, such as a pointer or handle to a data area you would like to associate with the window.

***wZoom* (rectangle).** This rectangle describes the size of the content region when the window is zoomed by clicking in its zoom box. Like those of any rectangle, this rectangle's coordinates are in top, left, bottom, right (TLBR) order. If you specify a zoom rectangle of (0,0,0,0), a default rectangle that describes the entire screen (except the menu bar) is used.

***wColor* (long).** This is a pointer to the window frame's color table. If this pointer is zero, a default color table is used.

***wOrigin* (point).** This is the position in the window's underlying document that corresponds to the top left-hand corner of the content region. This value is used to compute the positions of the thumbs in the right and bottom scroll bars. TaskMaster updates this value in the window record when a scrolling operation occurs. TaskMaster also uses SetOrigin to move the origin of the content region to this point before calling the routine to update the window (see below); the origin is reset to (0,0) on return from the update routine. If you draw inside a window outside of the update subroutine, first call StartDrawing to adjust the origin properly; use SetOrigin to return the origin to (0,0) after drawing.

***wData* (long).** This is the height (low word) and width (high word) of the entire document, measured in pixels. This data is needed to draw the right and bottom scroll bars properly. Set the height or width to 0 if you are not using right or bottom scroll controls, respectively.

***wMax* (long).** This is the maximum allowable height (low word) and width (high word) of the content region when resizing the window. TaskMaster passes this value to GrowWindow when a mouse-down event occurs in the window's grow box. Set it to 0 if you want to use the default value. which is the height and width of the desktop.

wScroll **(long).** This is the number of pixels to scroll the content region when the up or down arrow in the right scroll bar is selected (low word) and the number of pixels to scroll when the left or right arrow in the bottom scroll bar is selected (high word). These values are used by TaskMaster to handle activity in a scroll bar.

wPage **(long)** This is the number of pixels to scroll the content region when the up or down page region in the right scroll bar is selected (low word) and the number of pixels to scroll when the left or right page region in the bottom scroll bar is selected (high word). These values are used by TaskMaster to handle activity in a scroll bar.

wInfoRefCon **(long).** This is the reference constant passed to the information bar draw routine. This constant can be anything the application likes. It often holds a pointer to a line of text to be drawn in the information bar.

wInfoHeight **(word).** This is the height of the information bar.

wFrameDefProc **(long).** This is a pointer to the window's definition procedure. Set this to 0 for a standard document window.

wInfoDefProc **(long).** This is the address of the routine TaskMaster calls to draw the contents of the information bar. Such a routine must be installed if the wFrame byte indicates that the window has an information bar; if this routine is not installed, the system will crash when the window is drawn. The Window Manager pushes three parameters on the stack before calling the information bar subroutine with a JSL instruction: a pointer to the enclosing rectangle for the information bar, the wInfoRefCon value, and a pointer to the window. These parameters must be removed from the stack before the subroutine ends with an RTL instruction. Note that before the routine tries to access the application's direct page or data space, it should set the direct page and data bank registers accordingly (after saving the ones passed to the routine). On exit, the initial values for these registers must be restored.

wContDefProc **(long).** This is the address of the routine TaskMaster calls to draw the window's content region in response to an update event. Before TaskMaster calls the routine, it sets the local coordinates of the top left corner of the content region to the positions indicated by the values of the scroll controls (if applicable) and calls BeginUpdate. On entry, the direct page and data bank registers may not be the same as when TaskMaster was called, so the routine will have to change them to access the application's direct page and data areas. The routine must end with an RTL instruction. If wContDefProc is 0, or if TaskMaster is not used, the application itself must trap update events and handle them in an appropriate way.

***wPosition* (rectangle).** This rectangle, given in global coordinates, describes the initial position and size of the window's content region.

***wPlane* (long).** This is a pointer to the window behind which this window is to appear (long). If the pointer is -1, the window appears on top; if 0, the window appears at the bottom.

***wStorage* (long).** This is the address of the memory area to use for the window record (long). Set it to 0 if NewWindow is to allocate the memory space for you by calling NewHandle (the usual case).

THE INFORMATION BAR PROCEDURE

The general structure of an information bar procedure is shown in listing 6–2. In this particular example, a line of text is centered and drawn in the information bar. A pointer to the text string is in the wInfoRefCon parameter.

UPDATING A WINDOW

The wContDefProc (update) subroutine referred to in the NewWindow parameter list is important because it is responsible for ensuring that newly exposed portions of a window are properly filled in. Without such a subroutine, portions of the window will be erased when other windows are moved from in front of it or when the window is enlarged. The TextReader program shown later in this chapter is an example of how to write such a subroutine for a window containing lines of text.

If wContDefProc is 0, update events are passed through to the application for handling. (You should clear wContDefProc when designing New Desk Accessories because you cannot be sure that TaskMaster is being used by the main application; see chapter 9.) The update handler is similar to a wContDefProc subroutine but must include some extra window management chores. To handle updates, first call BeginUpdate:

```
PushLong WindowPtr      ;Pointer to window
_BeginUpdate
```

BeginUpdate saves the window's current visible region (which describes the portion of the window's content region you can actually see on the screen), sets the new visible region to the intersection of the old visible region and the update region, and then empties the update region.

After calling BeginUpdate, pass the window pointer to StartDrawing to select the window for drawing activity and to align the origin of the content region to the position reflected by the values of the thumbs in the scroll controls. This aligns the active portion of the document with the window's content region. Then redraw the portion of the window's document that appears within the new visible region. To do this, it is actually easier to redraw the entire content region of the window, because doing so avoids the problem of determining how to draw just an arbitrary portion of it. To do this, of course, you must always keep a record of exactly what is in the window. This can be the most difficult part of developing an application.

After the screen has been redrawn, call EndUpdate to restore the original visible region:

```
    PushLong WindowPtr        ;Pointer to window
    _EndUpdate
```

Next, set the origin of the window back to (0,0) with SetOrigin:

```
    PushWord #0
    PushWord #0
    _SetOrigin
```

The Window Manager does not work properly if you do not return the origin to (0,0) after updating the window.

CHANGING THE PROPERTIES OF A WINDOW

After you have created a window with NewWindow, you will need to change some of that window's properties in certain situations. For instance, you should alter the DataSize parameter to reflect a change in the size of the window's underlying document so that the scroll controls can be redrawn in proper proportion. To change such properties, you must update fields in the window record created from the NewWindow's parameter list, not in the parameter list itself. You do this with a group of Set functions, one for each of the named fields in the parameter list.

Table 6–2 contains a list of the relevant Set functions. When you change a parameter, the appearance of a window may not immediately change to reflect the value of the new parameter. To force the window to be redrawn, first hide the window and then make it visible again:

```
    PushLong WindowPtr
    _HideWindow               ;Remove window from screen
    PushLong WindowPtr
    _ShowWindow               ;Make the window visible
```

Table 6–2: Functions for Setting Window Parameters

Function	*Description*
SetWFrame	Sets the window frame bit vector
SetWTitle	Sets the title for the window
SetWRefCon	Sets the reference constant for the window
SetFullRect	Sets the zoom rectangle for the window
SetFrameColor	Sets the color table to be used with the window
SetContentOrigin	Sets the horizontal and vertical offsets to the data area corresponding to the top left-hand corner of the window
SetDataSize	Sets the width and height of the data area for a window
SetMaxGrow	Sets the maximum width and height for a window that can be resized
SetScroll	Sets the number of pixels to scroll one line horizontally and the number of pixels to scroll one line vertically
SetPage	Sets the number of pixels to scroll one page horizontally and the number of pixels to scroll one page vertically
SetInfoRefCon	Sets the parameter passed to the information bar drawing procedure
SetDefProc	Sets the address of the window definition procedure.
SetInfoDraw	Sets the address of the information bar drawing procedure
SetContentDraw	Sets the address of the window's update procedure

If there is more than one window on the screen, call SelectWindow after the code sequence above to make this window the active one.

There are also Get functions corresponding to each Set function. Use them to determine the current values of the window parameters. With some exceptions, Get functions are all called in the same general way:

1. Push space for result (word or long)
2. Push a pointer to the window record (long)
3. Call the Get function
4. Pop the result (the window parameter)

For a function dealing with strings, the result is a pointer to the string.

The exceptions relate to those functions dealing with rectangles, strings, and other data structures longer than four bytes. Instead of pushing space for a result, push a pointer to a space for the rectangle or string. The function returns the result in this space, so you do not have to pop a result from the stack.

REMOVING A WINDOW FROM THE DESKTOP

When you are finished with a window for good, use CloseWindow to free up the space reserved for it (if it was created with NewWindow), remove it from the window list, and erase it from the screen:

```
PushLong WindowPtr       ;Pointer to window
_CloseWindow
```

CloseWindow causes the window in the plane just below the window that was closed (if any) to be highlighted. It also generates an activate event for this window. Once a window is closed, you cannot use it again unless you redefine it with NewWindow.

If you simply want to erase a window from the screen without destroying its record, use HideWindow to make it invisible:

```
PushLong WindowPtr       ;Pointer to window
_HideWindow
```

To make the window visible again, pass its pointer to the ShowWindow function. You will also use ShowWindow to display a window for the first time if the visible bit in the window frame word is 0 when you call NewWindow.

Be aware that calling ShowWindow will not make the window active if there are other windows already on the screen. Use SelectWindow for that:

```
PushLong WindowPtr       ;Pointer to window
_SelectWindow            ;activate the window
```

SelectWindow removes the highlighting from the previously active window, brings the specified window to the front of the screen and highlights it, and then generates the appropriate activate events.

You will use SelectWindow more often to activate an inactive, but visible, window when GetNextEvent returns a mouse-down event that FindWindow says took place inside an inactive window. If you are using a TaskMaster—rather than a Get-

NextEvent—event loop, you will not actually have to call SelectWindow, because TaskMaster calls it for you.

To determine if the window in which an event took place is the active (frontmost) window, use FrontWindow:

```
PHA                         ;Space for result
PHA
_FrontWindow
PopLong  CurrentWindow  ;Pop the window pointer
```

If the result, CurrentWindow, matches the value of the window pointer that FindWindow returns, the activity did take place in the active window.

HANDLING MOUSE-DOWN ACTIVITY IN WINDOWS

The user-interface guidelines define the actions that should take place when various parts of a window are clicked. For example, a click in an inactive window should activate the window and bring it to the front of the screen. The Window Manager's TaskMaster function (introduced in chapter 5) performs many of the standard actions for you, so use it unless you need to deal with a mouse-down event in a different way. This section describes exactly how TaskMaster reacts to various events so that if you want to handle them on your own, you will know roughly what to do.

Here is a subroutine you could call to get a TaskMaster event to work with; the subroutine returns an event code in the accumulator:

```
EventLoop   PHA                    ;Space for result
            PushWord #$FFFF        ;Allow all events
            PushPtr TaskRecord ;Pointer to task record
            _TaskMaster
            PLA                    ;Get event code
            BEQ EventLoop          ;Loop if null event
            RTS

TaskRecord  ANOP
what        DS      2
message     DS      4
when        DS      4
where       DS      4
modifiers   DS      2
TaskData    DS      4                   ;TaskMaster data
TaskMask    DC      I4'$00001FFF' ;TaskMaster handles all
```

When TaskMaster detects a mouse-down event, it calls the FindWindow function to determine in which part of a window the event occurred. The location codes FindWindow can return are shown in table 6–3.

If FindWindow returns a $0000 event code (wNoHit), the event should be ignored.

Table 6–3: Location Codes Returned by FindWindow

Symbolic Name	*Code*	*Meaning*
wNoHit	$0000	Not in a window at all
wInDesk	$0010	In the desktop
wInMenuBar	$0011	In the system menu bar
wInContent	$0013	In the window's content region
wInDrag	$0014	In the window's drag region
wInGrow	$0015	In the window's size box
wInGoAway	$0016	In the window's close box
wInZoom	$0017	In the window's zoom box
wInInfo	$0018	In the window's information bar
wInSpecial	$0019	Special menu item selected (*)
wInDeskItem	$001A	Desk accessory item selected (*)
wInFrame	$001B	In any other part of the window
wInSysWindow	$8xxx	In desk accessory window

NOTE: In the instances marked by an asterisk, FindWindow does not actually return these codes; they are returned by TaskMaster if TaskMaster is not configured to deal with desk accessories selections (wInDeskItem) or if standard editing/closing menu items are used (wInSpecial).

Note that if the high-order bit of the location code is set to 1 (that is, the code is of the form $8xxx), the event occurred in a desk accessory. TaskMaster simply passes the event to the accessory by calling SystemClick. It then exits and returns a null event so that the application will not try to handle the event.

Other result codes refer to application windows and are handled in different ways by TaskMaster. These codes are described below.

wInMenuBar. TaskMaster handles activity in the menu bar by calling MenuSelect to determine which menu item was selected. See chapter 7 for more information on how TaskMaster behaves beyond this stage. After processing, TaskMaster exits and returns a null event code, wInMenuBar, wInDeskItem, or wInSpecial.

wInContent. If the window is not active, TaskMaster calls SelectWindow to make it active, exits, and returns a null event code. If it is active, TaskMaster puts a pointer to the window in TaskData, exits, and returns the wInContent code. It is then up to the application to handle the event in an appropriate way, by repositioning a cursor in a line of text or highlighting an icon, for example. If the interior of the window has controls like buttons and check boxes, you should use the Control Manager to see if they were clicked. Windows with controls are better handled by the Dialog Manager, as chapter 8 will discuss.

wInDrag. If the window is not active and the Open-Apple key is not down, TaskMaster calls SelectWindow to activate the window. It then calls DragWindow, which causes a dotted outline of the window to move around the screen as the mouse moves; when the button is released, the screen is redrawn at its new position. TaskMaster then exits and returns a null event code. If the window is active, or the Open-Apple key is down, SelectWindow is not called first.

wInGrow. TaskMaster calls GrowWindow, which causes a dotted outline of the window, anchored in the top left corner, to move in concert with the mouse. GrowWindow returns the new dimensions of the window, which TaskMaster passes to SizeWindow to redraw the window. TaskMaster then exits and returns a null event code.

wInGoAway. TaskMaster calls TrackGoAway to track the movement of the mouse until the button is released. If the button is released outside of the close box, TaskMaster exits and returns a null event code; otherwise, it puts the window pointer in TaskData, exits, and returns a wInGoAway event code. The application can then either permanently dispose of the window by calling CloseWindow or just temporarily remove it from the screen with HideWindow.

wInZoom. TaskMaster calls TrackZoom to track the movement of the mouse until the button is released. If the button is released outside of the zoom box, TaskMaster exits and returns a null event code. Otherwise, it calls ZoomWindow to resize the window to its zoomed size or, if the window has already been zoomed, to its prezoomed size; TaskMaster then exits and returns a null event code.

wInFrame. If the window is not active, TaskMaster calls SelectWindow to activate the window, exits, and returns a null event code. If the window is active and the event occurred in a scroll bar that is part of the window frame, TaskMaster calls TrackControl with a custom control action procedure. This action procedure performs necessary scrolling and updating of the window by calling the wContDefProc procedure that was defined when the window was created. TaskMaster then exits and returns a null event code. If the event occurred in some other part of the frame, TaskMaster exits and returns the wInFrame code.

wInDesk and wInInfo. TaskMaster does nothing special for these two types of events. It simply places the window pointer in TaskData and exits with the wInDesk or wInInfo code, as the case may be.

DRAWING IN A WINDOW

Objects are drawn in a window by moving a software-controlled, invisible pen around the screen with QuickDraw functions. These objects can be text characters, geometric objects, or random scribblings. The pen has several characteristics that can be set to affect the appearance of the object being drawn: an “ink” color and pattern, a size, and a transfer mode, for example.

Note that when you draw an object, only the bits for the pixels which fall inside the visible portion of the content region are actually transferred to the content region's memory area. Other pixels are ignored.

The functions discussed below affect only the active QuickDraw GrafPort, which is not necessarily the active window. To make a particular GrafPort active and to adjust the coordinate origin of the GrafPort properly, use StartDrawing:

```
    PushLong WindowPtr      ;Pointer to window
    _StartDrawing
```

You need to call StartDrawing only when you are drawing outside a wContDefProc window update subroutine.

If you do not do this, you may find that drawing operations take place in an unexpected window, or in the correct window but at an unexpected position. StartDrawing performs its function by calling SetPort to set the active drawing port and SetOrigin to set the origin to the position indicated by the values of the window's scroll controls.

Before drawing in a window, it is a good idea to save the pointer to the currently active drawing port with GetPort:

```
    PHA             ;space for result
    PHA
    _GetPort
    PopLong ThePort
```

When you are through drawing, you can restore the original port with SetPort:

```
    PushLong ThePort
    _SetPort
```

It is especially important that you do this in a multiwindow environment to ensure that you do not disrupt other drawing operations.

Two important data structures you will use quite often with QuickDraw functions are points and rectangles. A point is made up of a vertical position and a horizontal position, as follows:

```
MyPoint  DS  2           ;vertical position
         DS  2           ;horizontal position
```

Offsets to these two positions (from MyPoint) are given the symbolic names *v* (0) and *h* (2) in the STANDARD.ASM code module. So, for example, to load the horizontal position into the accumulator, you could use the following instruction:

```
LDA     MyPoint+h
```

A rectangle is made up of four words representing the position of the top row, the left side, the bottom row, and the right side:

```
MyRect   DS  2           ;top
         DS  2           ;left
         DS  2           ;bottom
         DS  2           ;right
```

The standard symbolic names for offsets to the four components are top (0), left (2), bottom (4), and right (6) and they are included in the STANDARD.ASM code module.

Note that you must change VidMode and XMaxClamp in the STANDARD.ASM file to the proper values for the super high-resolution mode you want to use. For 640-by-200 mode, set them to $80 and 640, respectively. For 320-by-200 mode, set them to $00 and 320.

PATTERNS AND COLORS

A pattern is an 8-by-8 pixel image that is transferred to the screen when lines are drawn or shapes are painted and filled. When a pattern is drawn in a window, it is blended with adjacent patterns to produce a smooth, regular pattern with no seam lines. A pattern can also be a solid block of color, enabling you to draw in colors other than black and white.

Patterns are actually transferred to the screen only after being processed by a mask. The mask is an 8-by-8 bit image (not a pixel image) that specifies which pixels in the pattern are to be drawn on the screen. Only those pixels in the 8-by-8 pixel image that correspond to 1 bits in the 8-by-8 bit mask image are drawn. This means that if the mask is all 1s, for example, the entire pattern is drawn to the screen.

The QuickDraw functions that deal with patterns are summarized in table 6–4.

A pattern is "chunky" in that each pixel is associated with either two color bits (640-by-200 mode) or four color bits (320-by-200 mode). Thus, it takes either 16 bytes or 32 bytes to define a pattern, depending on which graphics mode is active.

Table 6–4: QuickDraw Pattern-related Functions

Function	*Description*
SetPenPat	Sets the pen pattern to a specified pattern
GetPenPat	Returns the current pen pattern
SetSolidPenPat	Sets the pen pattern to a solid color
SetBackPat	Sets the background pattern to a specified pattern
GetBackPat	Returns the current background pattern
SetSolidBackPat	Sets the background pattern to a solid color
SolidPattern	Returns the pattern corresponding to a solid color
SetPenMask	Sets the pen mask to a specified mask
GetPenMask	Returns the current pen mask

For consistency, however, QuickDraw always allocates 32 bytes for a pattern in 640-by-200 mode; the second group of 16 bytes is simply a duplicate of the first group.

The default pen pattern is a solid black color. It can be changed using SetPenPat:

```
PushPtr ThePattern     ;Pointer to pattern
_SetPenPat
RTS

ThePattern 32I1'%01010101'      ;Pattern definition
```

A related function, GetPenPat, is called in the same way and fills the 32 bytes at ThePattern with the pattern definition.

If you want to set the pen pattern to a solid color so that you can draw in color, use SetSolidPenPat instead:

```
PushWord #ColorNum      ;Color number
_SetSolidPenPat
```

The colors associated with each color number were given in table 6–1 at the beginning of this chapter.

The difference between SetPenPat and SetSolidPenPat is that you pass a color number to SetSolidPenPat, not a pattern pointer. QuickDraw takes care of creating the appropriate pattern for you. The color number can be 0 to 15 for 320-by-200 mode or 0 to 3 for 640-by-200 mode.

If you want to determine what the pattern for a solid color looks like, use SolidPattern:

```
PushWord #ColorNum   ;color number
PushPtr ThePattern   ;Pointer to pattern space
_SolidPattern
RTS

ThePattern DS   32   ;Space for pattern
```

A background pattern is the pattern used to erase areas of a window or draw new areas that come into view. The default background pattern is solid white. It can be changed with SetBackPat and SetSolidBackPat, functions that are called in the same way as SetPenPat and SetSolidPenPat. There is also a GetBackPat function for determining what the current background pattern is.

SETTING UP THE DRAWING PEN

Before you start drawing text in a window, you must set the pen position. This is the location at which text drawing will begin. There are two QuickDraw functions for explicitly setting the pen position: MoveTo and Move.

MoveTo moves the pen position to a specific point in the local coordinate system of the window. To use it, push the horizontal and vertical components of the coordinate on the stack:

```
PushWord #2              ;Horizontal position
PushWord #10             ;Vertical position
_MoveTo
```

If you wish, you can combine the two PushWord operations into a single PushLong. This is handy if the point to which you want to move is already stored in memory. For example, suppose GetNextEvent or TaskMaster returns a button-down event in the content region of a window. To move the pen position to the point specified in the Where field of the event record, use the following subroutine:

```
PushLong Where
_GlobalToLocal           ;Switch to local coordinates

PushLong Where           ;This pushes h then v
_MoveTo
RTS
```

Notice that because TaskMaster and GetNextEvent return a global coordinate, that coordinate must be converted to a local coordinate for the currently active window before calling MoveTo.

When QuickDraw draws a text character, the current vertical pen coordinate becomes the baseline for the font. Because characters rise above the baseline, you should not specify a vertical coordinate of 0 for MoveTo; if you do, you will see only the descenders of the characters on the first line when you start drawing.

The Move function moves the pen location in a relative way. That is, with it you can move the pen so many pixels horizontally and vertically from the current pen location. For instance, if you want to drop down to the next text line in the window without changing the horizontal position (a line-feed operation), use the following code segment:

```
    PushWord #0                 ;Horizontal displacement
    PushWord #9                 ;Vertical displacement
    _Move
```

A value of 9 is used for the vertical displacement because this is the height of the system font. You can calculate this using the GetFontInfo function (as described below). Note that positive horizontal displacements are to the right and positive vertical displacements are downward.

To determine the current position of the pen, perhaps to see if you are still inside the window's content region, use GetPen:

```
            PushPtr ThePoint        ;Pointer to point
            _GetPen
            RTS

    ThePoint DS 4                   ;v, h position
```

GetPen returns the position in the local coordinates of the current GrafPort.

Setting the Pen State

You may want to set several other pen characteristics before beginning to draw graphics. (The functions described below do not affect text drawing operations.) The important characteristics are:

- The pen size
- The pen transfer mode
- The pen drawing pattern
- The pen mask

These four values are said to define the *pen state*. You can determine all their values at once using GetPenState:

```
              PushPtr PS_Record    ;Ptr to pen state record
              _GetPenState
              RTS
```

```
PS_Record ANOP
PnLoc     DS   4                   ;Pen position (point)
PnSize    DS   2                   ;Pen height
          DS   2                   ;Pen width
PnMode    DS   2                   ;Pen transfer mode
PnPat     DS   32                  ;Pen pattern
PnMask    DS   8                   ;Pen mask
```

The parameter table passed to GetPenState is called a pen state record.

To change the pen state, use the following functions:

- SetPenSize
- SetPenMode
- SetPenPat (described above)
- SetPenMask (described above)
- SetPenState

The size of the pen refers to the height and width of its nib. This nib hangs down and to the right of the current pen position. To change the pen size, say to 3 pixels by 5 pixels, use SetPenSize:

```
PushWord #3      ;New width in pixels
PushWord #5      ;New height in pixels
_SetPenSize
```

You can determine the current pen size with GetPenSize:

```
PushPtr PenSizeLoc  ;Pointer to result space
_GetPenSize
RTS

PenSizeLoc  DS  4
```

GetPenSize returns the width and height at PenSizeLoc.

The pen mode refers to the way in which QuickDraw combines the pattern flowing out of the pen with the pixels on the screen. The result of the combination is displayed on the screen. The different modes are summarized in table 6–5. (Note that some of these modes are for text drawing only.)

To change several pen characteristics at once, call SetPenState by passing it a pointer to a filled-in pen state record.

If you have changed several pen characteristics and you want to return to the standard pen state, call PenNormal (no parameters). It sets the pen size to 1 by 1, the pen mode to COPY, the pen pattern to black, and the pen mask to all 1s.

Table 6–5: The QuickDraw Transfer Modes

Transfer Mode	*Symbolic Name*	*Description*
$0000	COPY	Pixels are copied directly to th screen.
$0001	OR	Pixels are logically ORed with the screen pixels.
$0002	XOR	Pixels are logically exclusive-ORed with the screen pixels.
$0003	BIC	Pixel bits which are set to 1 cause corresponding pixels on the screen to be cleared to 0.
$0004	foreCOPY	*Text only:* Foreground pixels are copied directly to the screen.
$0005	foreOR	*Text only:* Foreground pixels are logically ORed with the screen pixels.
$0006	foreXOR	*Text only:* Foreground pixels are logically exclusive-ORed with the screen pixels.
$0007	foreBIC	*Text only:* Foreground pixel bits which are set to 1 cause corresponding pixel bits on the screen to be cleared to 0.
$8000	notCOPY	The pen pattern or text is inverted, then copied directly to the screen.
$8001	notOR	The pen pattern or text is inverted, then ORed with the screen pixels.
$8002	notXOR	The pen pattern or text is inverted, then logically exclusive-ORed with the screen pixels.

Table 6–5: Continued

Transfer Mode	*Symbolic Name*	*Description*
$8003	notBIC	The pen pattern or text is inverted, then those pixel bits which are set to 1 cause corresponding pixels on the screen to be cleared to 0.
$8004	notforeCOPY	*Text only:* The character rectangle is inverted, and the pixels that are on are copied directly to the screen.
$8005	notforeOR	*Text only:* The character rectangle is inverted, and the background pixels are logically ORed with the screen pixels.
$8006	notforeXOR	*Text only:* The character rectangle is inverted, and the background pixels are logically exclusive-ORed with the screen pixels.
$8007	notforeBIC	*Text only:* The character rectangle is inverted, and the background pixels which are set to 1 cause corresponding pixel bits on the screen to be cleared to 0.

FONT CHARACTERISTICS

A font is a set of characters having the same general ornamental design. One font, called the system font, is stored in the GS ROM and is always available for use by the QuickDraw text-drawing functions. Other fonts may be defined and stored in the SYSTEM/FONTS/ subdirectory on the boot disk. They can be loaded into memory and made available to an application using the Font Manager. QuickDraw can draw characters in any available font, but it will use the system font unless you specify otherwise.

Although QuickDraw has a few font-related functions, you should use the more powerful Font Manager functions instead. To start up the Font Manager, use FMStartup:

```
PushWord MyID            ;Program ID (from MMStartup)
PushWord DPAddr          ;Address of one page in bank $00
_FMStartup
```

The Font Manager requires the presence of most GS tool sets, including the rarely-used List Manager. Make sure they are all loaded before calling FMStartup.

FMStartup scans the SYSTEM/FONTS/ subdirectory of the boot disk and makes a list of the font families it finds there. A font family is a general font type such as Helvetica, Courier, or Venice. The default system font is called Shaston.

A font of a particular family, size, and style is described by a fontID long word. Here is its structure:

- bits 00-15 : font family number
- bits 16-23 : font style byte
- bits 24-31 : font size byte

The family number is the number the Font Manager assigns to the font when it puts the font in its font list. Bits in the style byte affect the style of the font (bold, italic, underline, outline, or shadow); style bytes are described below in the discussion of SetTextFace. The font size is the point size of the font (1 point = 1/72 of an inch).

The user can install a new font relatively easily, at least if the application calls ChooseFont to bring up a standard font selection dialog box:

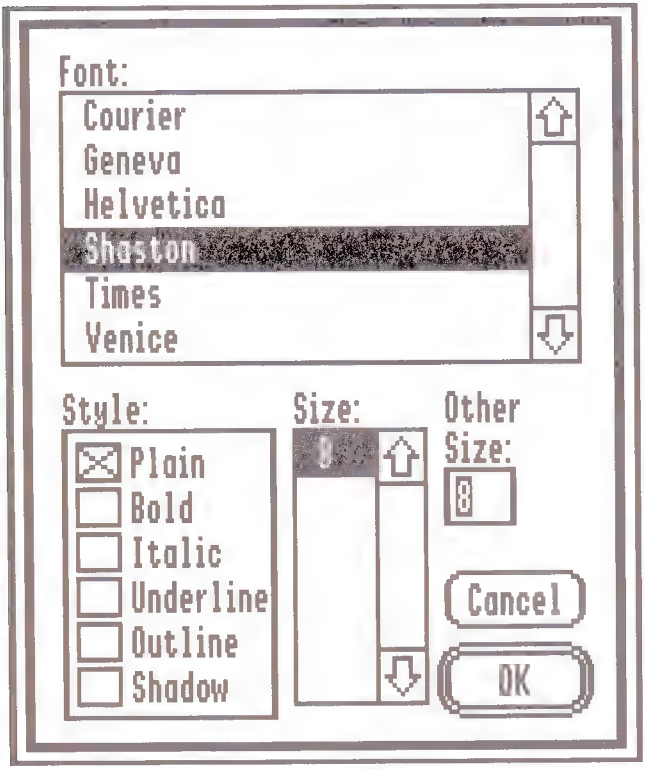
```
PHA                      ;space for result
PHA
PushLong TheFontID       ;fontID of current font
PushWord #0              ;0 = list all fonts
_ChooseFont
PopLong NewFontID        ;Result is new fontID
```

Push zero instead of TheFontID to select the system font.

The form of the font selection dialog box is shown in figure 6–8. It consists of a scrollable window that holds a list of font names for selection, checkboxes for selecting the font style, and a scrollable window that holds a list of point sizes that are available. There is also an edit box for typing in any other point size to which a font should be scaled; scaled fonts can look ragged, however.

ChooseFont returns the fontID for the font selected, or 0 if the dialog box was canceled.

Figure 6–8. The Font-selection Dialog Box



Font:

- Courier
- Geneva
- Helvetica
- Shaston
- Times
- Venice

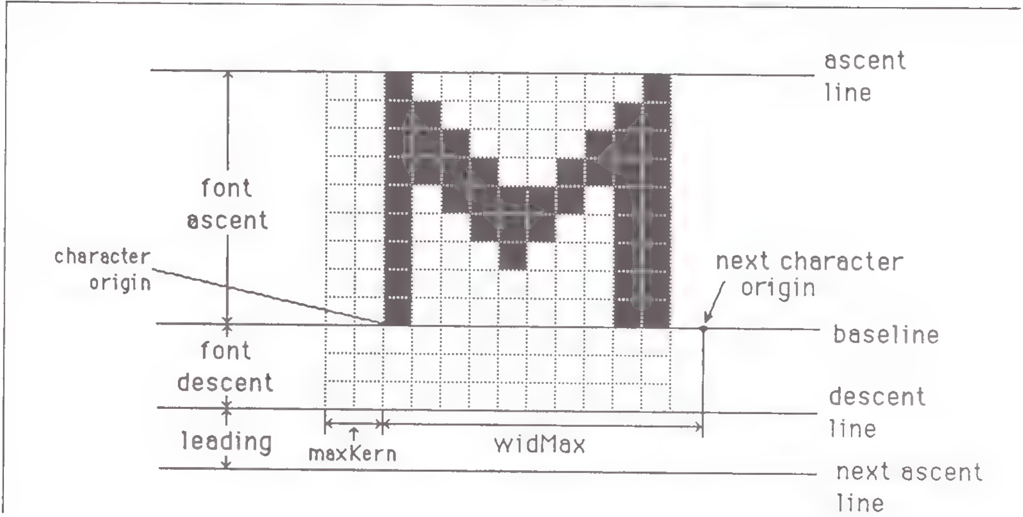
Style:

☒ Plain
□ Bold
□ Italic
□ Underline
□ Outline
□ Shadow

Size: 8

Other Size: 8

Figure 6–9. The Characteristics of an Apple IIGS Font



To make the selected font the current font, so that it will be used by QuickDraw text drawing functions, call InstallFont:

```
PushLong NewFontID          ;fontID of desired font
PushWord #0                 ;0 = enable scaling
_InstallFont
```

The second parameter controls whether font scaling is enabled. If bit 0 is 1, scaling is not enabled.

When you are through using the new font, you can make the system font the current font again with the LoadSysFont function (no parameters). If you are never going to use the new font again, make it purgeable with SetPurgeStat:

```
PushLong NewFontID          ;fontID of font
PushWord #%00010000         ;bit 4 = 1 (purge)
_SetPurgeStat
```

Bit 4 of the second parameter controls whether the specified font is to be purgeable (1) or not purgeable (0).

Font Definitions

As shown in figure 6–9, each character in a font is defined inside a rectangular grid, called the *font rectangle*, which encloses the largest character in the font. Another rectangle, the *character rectangle*, is the smallest rectangle that can enclose the outline of the character. The size of this rectangle will vary, depending on the size of the character.

Those points in a character rectangle that are highlighted are called foreground points, because they define the actual shape of the character. The other points are background points. When a character is transferred to a GrafPort, the foreground points become black pixels and the background points become white pixels, although you can change these default colors with SetBackColor and SetForeColor if you wish. The colors are defined by either a 2-bit number (640-by-200 mode) or a 4-bit number (320-by-200 mode), so there are either four or sixteen colors to choose from.

Each character in a font is defined relative to two reference points, the *baseline* and the *character origin*. The baseline is an imaginary line on which the character rests; it serves much the same purpose as a line on a piece of notepaper. The number of pixel lines between the baseline and the top of the font rectangle is the *ascent*. The number of lines between the baseline and the bottom of the font rectangle is the *descent*—this is where the descenders of characters such as g, y, j, p, and q appear.

The character origin marks the position in the character rectangle that is aligned with the pen position before the character is drawn. The widMax of a font is the maximum number of pixels between the character origins of two successive characters. For a proportional font, such as the system font, the actual widths of most characters are less than this. You can, however, tell QuickDraw to draw system font characters that are all widMax pixels wide using SetFontFlags:

```
PushWord #1      ;1 = fixed-width, 0 =proportional
_SetFontFlags
```

To return to proportional mode, pass a 0 to SetFontFlags. You probably will not use fixed-width characters very often, however, because the gaps between characters are large and unsightly. You also cannot fit as many characters across the width of the screen.

The leading of a font is defined as the number of blank lines between the descent line and the bottom of the font rectangle. This area acts as a spacer between successive lines of text. GetFontInfo returns the ascent, descent, widMax, and leading of the currently active font in a four-word font information record. For the system font, the results are as follows:

- ascent 7
- descent 1
- widMax 12
- leading 1

The subroutine below, which incorporates GetFontInfo, can be used to move the pen position to the next line in the window:

```
v           GEQU     0              ;vertical field offset
CRLF        START
            PushPtr FIRecord        ;Pointer to font record
            _GetFontInfo
            CLC
            LDA      Ascent
            ADC      Descent
            ADC      Leading
            STA      FontHeight

            PushPtr PenPos          ;Pointer to a point
            _GetPen                 ;Get current pen position

            CLC
            LDA      PenPos+v       ;Get current vertical
            ADC      FontHeight     ;Add line height

            PEA      2              ;horizontal (left edge)
            PHA                     ;vertical (next line)
            _MoveTo
            RTS

FIRecord    ANOP                    ;Font information record
Ascent      DS       2
Descent     DS       2
WidMax      DS       2
Leading     DS       2
FontHeight  DS       2

PenPos      DS       4              ;point (v,h)

            END
```

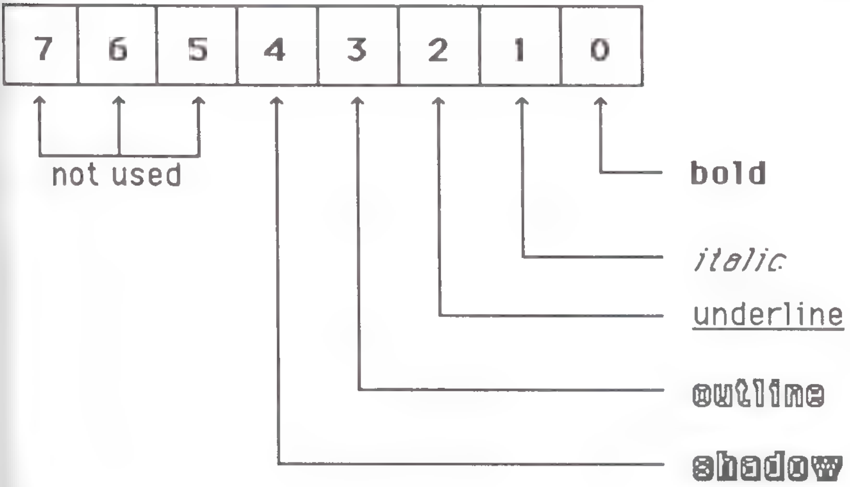
This subroutine adds ascent, descent, and leading together to calculate the height of the font. It then adds the height to the current vertical pen position to determine the vertical position of the next baseline. Finally, it calls MoveTo to position the pen at the left side of this line.

You can assign certain typeface attributes to a font using the SetTextFace function. The five attributes currently defined are bold, italic, underline, outline, or shadow or any combination of the five. Bits in the word parameter passed to SetTextFace correspond to attributes as shown in figure 6–10. If all the bits are 0, the typeface is called plain.

For example, to switch the typeface to bold, execute the following code segment:

```
PushWord #%00000001      ;textface word
_SetTextFace
```

Figure 6–10. The Style Byte Used by SetTextFace



Note that some fonts, including the system font, may not support all of these attributes.

GetTextFace returns the current textface word:

```
PHA                         ;space for result
_GetTextFace
PLA                         ;Get the result
```

DRAWING CHARACTERS

To print a single character in a window, position the pen with Move or MoveTo. Then push the ASCII code for the character on the stack and call DrawChar:

```
PushWord #$41               ;ASCII code for "A"
_DrawChar
```

As in all QuickDraw text drawing functions, the ASCII code for standard characters must have bit 7 cleared to 0. See appendix 1 for a description of the ASCII encoding scheme. The system font assigns various foreign characters, monetary symbols, and common icons to the 128 codes that have bit 7 set to 1.

There are three other "Draw" functions for drawing sequences of characters stored in memory: DrawString, DrawCString, and DrawText. The one to use depends on whether the sequence begins with a length byte (a Pascal-type string), ends with a zero byte (a C-type string), or is an arbitrary sequence of characters with no length indicator (text).

Three subroutines showing how to use these drawing functions are given below:

```
DrawSub1    PushPtr MyPString             ;Pointer to string
            _DrawString
            RTS


DrawSub1    PushPtr MyCString             ;Pointer to string
            _DrawCString
            RTS


DrawSub2    PushPtr MyText                ;Pointer to string
            PushWord #14                  ;length
            _DrawText
            RTS


MyPString   DC    I1'16'                  ;length
            DC    C'A Pascal string.'     ;the characters


MyCString   DC    C'A C string.'          ;the characters
            DC    I1'0'                   ;trailing 00


MyText      DC    C'This is a text string.'
```

Notice that for DrawText, you identify the portion of the text to be displayed by specifying a starting position and the number of characters with which you want to work. This is not necessary for DrawString or DrawCString, because the length is known and the entire string is always printed.

If you use Pascal-type strings quite often, you will find it convenient to define them with a macro called STR. STR is useful because it automatically calculates the length of the string and inserts it before the string's character codes. You do not have to count the characters yourself. Here is how to define the string "A Pascal string" using STR instead of two DC directives:

```
MyPString STR 'A Pascal string'
```

A definition for the STR macro is given in listing 6–3.

It is important to realize that the pen position automatically advances along the line as you draw characters. There is no need to set it explicitly with Move or MoveTo unless you want to move to another line or to a non-adjacent position on the same line.

Note that none of the text-drawing functions react to control characters in standard ways. If you pass a carriage return code ($0D) to DrawChar, for example, the drawing position does not automatically move to the left side of the next line in a window; instead, an inverse question mark character appears on the screen (indicating that there is no symbol for the character in the font definition). It is up to the application to intercept control characters and handle them appropriately.

Text Measuring

It is often convenient to know the width of a character, or a series of characters, before drawing. Knowing the width, you can easily center the text or determine if there will be enough room to display it on the current line. The program in listing 6–4 shows how to center a line of text in a window, for example.

QuickDraw has a set of text measuring functions you can use to determine widths:

- CharWidth (for a single character)
- TextWidth (for a sequence of characters)
- StringWidth (for a string preceded by a length byte)
- CStringWidth (for a string followed by a $00 byte)

To use these functions, first push a dummy word on the stack to reserve space for the result (the width in pixels), and then push the ASCII code for the character (CharWidth) or a pointer to the text or string (TextWidth, StringWidth, CStringWidth). For TextWidth, you must also push the length of the text string. Finally, call the function and pop the result from the stack.

Text Color

Using SetForeColor and SetBackColor, you can set the foreground and background color of text you draw in a window. Both require a parameter word that contains the color number to which you want to switch. In 640-by-200 mode, this number can be 0 to 3; for 320-by-200 mode, it can be 0 to 15.

Here is how to use SetForeColor and SetBackColor:

```
PushWord #ColorNum        ;Push new color number
_SetForeColor             ;(or SetBackColor)
```

To determine the current color characteristics of the font, use GetForeColor and GetBackColor. They return the color number on the stack:

```
    PHA                       ;space for result
    _GetBackColor
    PLA
    STA     TheColor          ;pop the result
```

The default background and foreground colors are white and black, respectively.

The program in listing 6–5 shows how to display text in all 16 colors in 320-by-200 mode using SetForeColor.

Transfer Modes for Text

A transfer mode refers to the technique QuickDraw uses to draw a text character on the screen. The mode defines how QuickDraw is to combine a pixel in the character with the corresponding pixel in the screen buffer.

The possible transfer modes were described in table 6–5. The default mode is COPY, which causes the pixels in the character rectangle to replace the corresponding pixels on the screen.

Another particularly useful mode is foreXOR. If you use this mode, you can draw text on any background, and then erase the text and redraw the background simply by redrawing the text again at the same position.

To set the transfer mode, use SetTextMode:

```
    PushWord #TextMode        ;Text transfer mode code
    _SetTextMode
```

You can determine what the currently active mode is using GetTextMode:

```
    PHA                       ;space for result
    _GetTextMode
    PLA
    STA     TheMode           ;Pop the mode code
```

It is hard to visualize the effect of some of the transfer modes. It is best to write a short program that uses different modes to draw text on differently colored backgrounds.

DRAWING LINES AND SHAPES

QuickDraw has several functions for drawing objects other than characters or text strings. You can use these functions to draw lines, rectangles, arbitrary regions, polygons, ovals, round rectangles, and arcs.

Lines

You can draw straight lines with the Line and LineTo functions. LineTo draws a line from the current pen location to the specified point:

```
PushWord #43            ;Horizontal position
PushWord #22            ;Vertical position
_LineTo                 ;Draw the line
```

With Line you can draw a line that terminates at a position that is relative to the current pen position:

```
PushWord #10            ;Horizontal displacement
PushWord #5             ;Vertical displacement
_Line
```

Positive horizontal displacements move the pen to the right. Positive vertical displacements move it down.

After either Line or LineTo is called, the pen position moves to the end of the new line.

Shapes

QuickDraw uses five fundamental drawing operations to manipulate the shapes it supports:

Framing—drawing an outline of the shape.

Painting—filling the interior of a shape with the current pen pattern. You can set the pattern to a particular color using SetSolidPenPat or to any arbitrary pattern using SetPenPat.

Erasing—replacing the interior of a shape with the background pattern. You can set the background pattern using SetSolidBackPat or SetBackPat.

Inverting—inverting the pixels inside a shape.

Filling—filling the interior of a shape with a specified pattern.

The specific shape-drawing functions for objects are summarized in table 6–6.

Table 6–6: QuickDraw II Shape-drawing Functions

Function	*Description*
FrameRect	Draws the outline of a rectangle
FrameRgn	Draws the outline of a region
FramePoly	Draws the outline of a polygon
FrameOval	Draws the outline of an oval
FrameRRect	Draws the outline of a round rectangle
FrameArc	Draws the outline of an arc
PaintRect	Fills the interior of a rectangle with the current pen pattern
PaintRgn	Fills the interior of a region with the current pen pattern
PaintPoly	Fills the interior of a polygon with the current pen pattern
PaintOval	Fills the interior of an oval with the current pen pattern
PaintRRect	Fills the interior of a round rectangle with the current pen pattern
PaintArc	Fills the interior of an arc (a wedge) with the current pen pattern
EraseRect	Fills the interior of a rectangle with the current background pattern
EraseRgn	Fills the interior of a region with the current background pattern
ErasePoly	Fills the interior of a polygon with the current background pattern
EraseOval	Fills the interior of an oval with the current background pattern
EraseRRect	Fills the interior of a round rectangle with the current background pattern
EraseArc	Fills the interior of an arc (a wedge) with the current background pattern

Table 6–6: Continued

Function	*Description*
InvertRect	Inverts the pixels in a rectangle
InvertRgn	Inverts the pixels in a region
InvertPoly	Inverts the pixels in a polygon
InvertOval	Inverts the pixels in an oval
InvertRRect	Inverts the pixels in a round rectangle
InvertArc	Inverts the pixels in an arc (The arc is defined by a starting angle and an angular extent.)
FillRect	Fills the interior of a rectangle with a specified pattern
FillRgn	Fills the interior of a region with a specified pattern
FillPoly	Fills the interior of a polygon with a specified pattern
FillOval	Fills the interior of an oval with a specified pattern
FillRRect	Fills the interior of a round rectangle with a specified pattern (The curvature of the corners can also be set.)
FillArc	Fills the interior of an arc with a specified pattern (The arc is defined by a starting angle and an angular extent.)

To draw the outline of a rectangle in a window, call FrameRect by passing a pointer to the definition of the rectangle:

```
          PushPtr TheRect       ;Pointer to rectangle
          _FrameRect
          RTS

   TheRect DC     I2'30,20,100,120' ;Top, left, bottom, right
```

Notice that the coordinates for the rectangle are specified in top, left, bottom, right (TLBR) order.

To erase the rectangle, use EraseRect:

```
PushPtr TheRect         ;Pointer to rectangle
_EraseRect
```

To invert the pixels inside the rectangle, use InvertRect:

```
PushPtr TheRect         ;Pointer to rectangle
_InvertRect
```

To fill the interior of a rectangle with the current pen pattern, use PaintRect:

```
PushPtr TheRect           ;Pointer to rectangle
_PaintRect
```

To fill it with any arbitrary pattern, use FillRect:

```
PushPtr TheRect           ;Pointer to rectangle
PushPtr ThePattern        ;Pointer to pattern definition
_FillRect
```

The pattern definition defines an 8 by 8 pixel grid, as explained in the section above called "Patterns and Colors."

The corresponding shape-drawing functions for other shapes have suffixes of Rgn (regions), Poly (polygons), Oval (ovals), RRect (round rectangles), and Arc (arcs). The functions are called in much the same way as the corresponding rectangle functions, but instead of pushing a pointer to a rectangle, push the following information:

For regions, a handle to the region

For polygons, a handle to the polygon definition

For ovals, a pointer to the imaginary rectangle enclosing the oval

For round rectangles, a pointer to the imaginary rectangle enclosing the round rectangle, followed by two words describing the width and height of the imaginary oval inscribed in each corner

For arcs, a pointer to the imaginary rectangle enclosing the imaginary oval of which the arc forms a part, followed by two words describing the starting angle of the arc (in degrees) and the extent of the arc (in degrees). Angles are measured relative to a vertical axis moving up from the center of the oval

Note that the interior of an arc is actually the wedge bounded by the arc and the imaginary lines from the ends of the arc to the center of the enclosing oval.

Polygons

In the previous section, we came across a shape called a polygon. A polygon, as any student of geometry can tell you, is an enclosed, multisided object. To define a polygon with QuickDraw, first open a polygon record by calling OpenPoly, as follows:

```
PHA                         ;Space for result (handle)
PHA
_OpenPoly
PopLong PolyHndl            ;Pop handle to polygon
```

The next step is to draw the outline of the polygon by making a series of calls to LineTo until the entire outline has been traversed. These lines are not actually displayed on the screen; the LineTo parameters are just stored in the polygon record. Finally, call ClosePoly (no parameters) to complete the polygon creation process.

Using the polygon handle returned by OpenPoly, you can perform all the standard QuickDraw drawing operations, such as FramePoly, PaintPoly, and FillPoly. The polygon you draw appears at the same position at which it was defined, unless you shift its position with the OffsetPoly function:

```
PushLong PolyHndl           ;Polygon handle
PushWord #horiz             ;Horizontal displacement
PushWord #vert              ;Vertical displacement
_OffsetPoly
```

Positive displacements are down and to the right.

When you do not need the polygon any more, dispose of it by calling KillPoly:

```
PushLong PolyHndl           ;Handle to polygon record
_KillPoly
```

This frees up the memory space QuickDraw previously allocated for the polygon record.

An example of how to deal with polygons is given in listing 6–6. This code defines a pentagon and draws it on the screen.

CREATING AN APPLICATION WITH WINDOWS

Now that methods of creating windows and drawing characters and objects have been covered, it is time to put it all together. The program in listing 6–7 shows how to implement many of the important functions that have been discussed in this chapter. It also illustrates how to deal with scroll controls and update procedures.

This program, called TextReader, loads a readable text file into memory from disk and lets you view any part of it using the right and bottom scroll controls of a window.

In this program, the file is loaded into memory using several ProDOS 16 commands that have not been covered yet—OPEN, NEWLINE, GET_EOF, READ, and CLOSE. These commands will be discussed in chapter 10, although you can probably guess what most of them do already. The program also uses an SFGetFile dialog box to request the name of the file to be loaded. This dialog box will also be discussed in chapter 10.

After loading a file from disk, TextReader uses NewWindow to create a window with scroll controls in its window frame. It does this by setting the right and bottom scroll bits of the window frame word in NewWindow's parameter list. It also sets the bits for a title bar, zoom box, and grow box.

Several other items in the window parameter list must be set up for any window containing scroll controls. First, you must specify the number of lines the content region of the window should scroll when the mouse is pressed in different parts of the scroll controls (except the thumbs). TextReader is designed to scroll one entire line of text if the mouse is pressed in the up or down arrow of the right scroll control. Thus, a value of 9, the pixel height of the system font TextReader uses, is stored in the appropriate position in the parameter table. A convenient scrolling distance for activity in the left or right arrow of the bottom scroll bar is 12 pixels, because this is the maximum width of a character in the system font.

The scrolling distance for activity in the page-up or page-down regions of the right scroll bar is the full height of the content region, minus the height of one line (9 pixels). The height of one line is subtracted to ensure that all lines are not scrolled off the screen—this makes it easier for the user to keep track of a scrolling operation. Similarly, the scrolling distance for the page-left and page-right regions of the bottom scroll bar is set to the width of the content region, minus 12. Both of these distances are updated every time TaskMaster returns a non-null event in case the window has been resized by zooming or by dragging the grow box.

The next step is to specify the maximum height and width, in pixels, of the block of text being displayed in the window. This information is needed so that the size of the scroll control thumbs and page regions can be calculated. For example, the ratio of the height of the thumb in the right scroll bar to the height of the entire scroll bar is the same as the ratio of the height of the content region to the height of the document.

In TextReader, the maximum width is set to 960 on the assumption that no line will be wider than 80 characters (remember, the maximum width of a single character is 12 pixels). The maximum height is set initially to an arbitrary 1,024, but this is changed once the file has been loaded and its height is known. TextReader calculates the height by scanning the loaded file for end-of-line characters (carriage returns) and adding 9 to the DocSize variable for every one it finds. An arbitrary figure of 20 pixels is added to the height to give the document a bottom margin.

Another parameter you must set before calling NewWindow is the one holding the position within the document that corresponds to the top left corner of the content region. NewWindow uses this parameter to determine where to draw the thumbs of the scroll bars. TextReader sets this position to 0,0 so that the top line of the text will appear at the top of the window when the content region is updated for the first time.

The last scrolling-related parameter that must be set is the one containing the address of the subroutine that TaskMaster calls when the window needs updating. Every scrolling operation forces a screen update, because every such movement brings a new portion of the document into view. This subroutine is called WindUpdate in the TextReader program and is responsible for redrawing the content region of the window.

Before TaskMaster calls the WindUpdate screen updating subroutine, it uses SetOrigin to assign the top left-hand corner of the window's content region to the position within the document given by the values of the scroll controls. This means that the content region is already aligned with the active portion of the document.

The easiest way to update the content region is simply to redraw the entire document; because drawing is clipped to the content region, you will not mess up other areas of the screen. This is not a satisfactory method for large documents, however, because it is very slow—after all, you must go through the charade of redrawing portions of the document that do not actually appear in the window.

A much faster alternative, the one used by TextReader, is to redraw only that portion of the document appearing in the content region. To locate the first line that will appear in the content region, TextReader first zeroes a counter, then adds 9 (the line height) to the counter for each end-of-line character (carriage return) it finds while scanning through the document from the beginning. When the counter becomes greater than the vertical position of the top of the content region (stored at PortRect+top), the search is over. Each line in the document is then drawn (with DrawText) until the pen position is one clear line below the bottom of the content region. (Carriage return codes are handled by calling CRLF to move the invisible drawing pen to the left side of the next line.) TextReader does not bother to do range checking on the sides, although you could add code to do this to make updates happen a bit faster.

TaskMaster calls the update handler with a JSL instruction, so the handler ends with an RTL instruction. Do not end with an RTS instruction as you would if the routine was called with JSR. Another tip on writing an update handler relates to the direct page: the active direct page when TaskMaster calls the update handler is not the direct page the application uses. If you must access data in the application's direct page, perhaps to use pointers you have previously set up in direct page, switch to the direct page first. The WindUpdate subroutine does this so that it can access the direct page pointer to the text block.

Once TextReader defines the window and loads the text file, it enters a short TaskMaster event loop. Because TaskMaster automatically takes care of all mouse-down events in the scroll bars and handles all update events, the application does not have to do anything special to support scrolling.

REFERENCE SECTION

Table R6–1: The Major Functions in the QuickDraw II Tool Set ($04)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
CharWidth	$A8	result (W)	Width of character in pixels
		Char (W)	The character
ClosePoly	$C2	[no parameters]	
CStringWidth	$AA	result (W)	Width of string in pixels
		TheCString (L)	Ptr to C-style text string
DrawChar	$A4	Char (W)	ASCII code for character
DrawCString	$A6	TheCString (L)	Ptr to C-style text string
DrawString	$A5	TheString (L)	Ptr to text string
DrawText	$A7	TheText (L)	Ptr to start of text
		Count (W)	Number of characters to draw
EraseArc	$64	Rectangle (L)	Ptr to rectangle
		StartAngle (W)	Starting angle
		ArcAngle (W)	Extent of angle
EraseOval	$5A	Rectangle (L)	Ptr to rectangle

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
ErasePoly	$BE	PolyHndl (L)	Handle to polygon
EraseRect	$55	Rectangle (L)	Ptr to rectangle
EraseRgn	$7B	RgnHndl (L)	Handle to region
EraseRRect	$5F	Rectangle (L)	Ptr to rectangle
		OvalWidth (W)	Width of corner oval
		OvalHeight (W)	Height of corner oval
FillArc	$66	Rectangle (L)	Ptr to arc's rectangle
		StartAngle (W)	Starting angle
		ArcAngle (W)	Extent of angle
		PatternPtr (L)	Ptr to fill pattern
FillOval	$5C	Rectangle (L)	Ptr to oval's rectangle
		PatternPtr (L)	Ptr to fill pattern
FillPoly	$C0	PolyHndl (L)	Handle to polygon
		PatternPtr (L)	Ptr to fill pattern
FillRect	$57	Rectangle (L)	Ptr to rectangle
		PatternPtr (L)	Ptr to the fill pattern
FillRgn	$7D	RgnHndl (L)	Handle to region
		PatternPtr (L)	Ptr to the fill pattern
FillRRect	$61	Rectangle (L)	Ptr to rrect's rectangle
		OvalWidth (W)	Width of corner oval
		OvalHeight (W)	Height of corner oval
		PatternPtr (L)	Ptr to fill pattern
FrameArc	$62	Rectangle (L)	Ptr to rectangle
		StartAngle (W)	Starting angle
		ArcAngle (W)	Extent of angle
FrameOval	$58	Rectangle (L)	Ptr to rectangle
FramePoly	$BC	PolyHndl (L)	Handle to polygon

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
FrameRect	$53	Rectangle (L)	Ptr to rectangle
FrameRgn	$79	RgnHndl (L)	Handle to region
FrameRRect	$5D	Rectangle (L)	Ptr to rectangle
		OvalWidth (W)	Width of corner oval
		OvalHeight (W)	Height of corner oval
GetBackColor	$A3	result (W)	Background color number
GetBackPat	$35	PatternPtr (L)	Ptr to the background pattern
GetColorEntry	$11	result (W)	Color entry value
		TableNum (W)	Color table number
		EntryNum (W)	Color number
GetColorTable	$0F	TableNum (W)	Color table number
		ColorTblPtr (L)	Ptr to space for color table
GetFontInfo	$96	FIRectPtr (L)	Ptr to font info record
GetForeColor	$A1	result (W)	Foreground color number
GetPen	$29	PointPtr (L)	Ptr to space for point result
GetPenMask	$33	MaskPtr (L)	Ptr to space for pen mask
GetPenMode	$2F	result (W)	Current pen mode
GetPenPat	$31	PatternPtr (L)	Ptr to current pen pattern
GetPenSize	$2D	PointPtr (L)	Ptr to width/height result
GetPenState	$2B	PenStatePtr (L)	Ptr to space for PS record
GetPort	$1C	result (L)	Ptr to current GrafPort
GetPortRect	$20	RectPtr (L)	Ptr to space for rect result
GetSCB	$13	result (W)	Scanline control byte
		ScanLine (W)	Scanline number
GetTextFace	$9B	result (W)	Current textface word
GetTextMode	$9D	result (W)	Current text drawing mode
GlobalToLocal	$85	PointPtr (L)	Ptr to point to convert

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
InitColorTable	$0D	ColorTblPtr (L)	Ptr to space for color table
InvertArc	$65	Rectangle (L)	Ptr to arc's rectangle
		StartAngle (W)	Starting angle
		ArcAngle (W)	Extent of angle
InvertOval	$5B	Rectangle (L)	Ptr to oval's rectangle
InvertPoly	$BF	PolyHndl (L)	Handle to polygon
InvertRect	$56	Rectangle (L)	Ptr to rectangle
InvertRgn	$7C	RgnHndl (L)	Handle to region
InvertRRect	$60	Rectangle (L)	Ptr to rrect's rectangle
		OvalWidth (W)	Width of corner oval
		OvalHeight (W)	Height of corner oval
KillPoly	$C3	PolyHndl	Handle to polygon
Line	$3D	hOffset (W)	Horizontal offset
		vOffset (W)	Vertical offset
LineTo	$3C	hPos (W)	Horizontal position
		vPos (W)	Vertical position
LocalToGlobal	$84	PointPtr (L)	Ptr to point to convert
Move	$3A	DispX (W)	Horizontal displacement
		DispY (W)	Vertical displacement
MoveTo	$3B	HorizPos (W)	Horizontal position
		VertPos (W)	Vertical position
OffsetPoly	$C4	PolyHndl (L)	Handle to polygon
		hOffset (W)	Horizontal displacement
		vOffset (W)	Vertical displacement
OpenPoly	$C1	result (L)	Handle to polygon

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
PaintArc	$63	Rectangle (L)	Ptr to rectangle
		StartAngle (W)	Starting angle
		ArcAngle (W)	Extent of angle
PaintOval	$59	Rectangle (L)	Ptr to rectangle
PaintPoly	$BD	PolyHndl (L)	Handle to polygon
PaintRect	$54	Rectangle (L)	Ptr to rectangle
PaintRgn	$7A	RgnHndl (L)	Handle to region
PaintRRect	$5E	Rectangle (L)	Ptr to rectangle
		OvalWidth (W)	Width of corner oval
		OvalHeight (W)	Height of corner oval
PenNormal	$36	[no parameters]	
QDShutDown	$03	[no parameters]	
QDStartup	$02	DPAddr (W)	Address of 3 pages in bank 0
		MasterSCB (W)	$8000 = 640/$0000 = 320
		MaxWidth (W)	Size of largest pixel map
		UserID (W)	ID tag for memory allocation
SetAllSCBs	$14	NewSCB (W)	New SCB value for all lines
SetBackColor	$A2	ColorNum (W)	Background color number
SetBackPat	$34	PatternPtr (L)	Ptr to new background pattern
SetColorEntry	$10	TableNumber (W)	Color table number (0-15)
		EntryNumber (W)	Color number in table (0-15)
		NewColor (W)	New color value
SetColorTable	$0E	TableNumber (W)	Color table number (0-15)
		ColorTblPtr (L)	Ptr to new color table

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
SetFontFlags	$98	FontFlag (W)	New font flags
SetForeColor	$A0	ColorNum (W)	Foreground color number
SetOrigin	$23	xOrigin (W)	X coord of upper-left corner
		yOrigin (W)	Y coord of upper-left corner
SetPenMask	$32	MaskPtr (L)	Ptr to the new pen mask
SetPenMode	$2E	PenMode (W)	New pen mode
SetPenPat	$30	PatternPtr (L)	Ptr to new pen pattern
SetPenSize	$2C	PenWidth (W)	Width of pen in pixels
		PenHeight (W)	Height of pen in pixels
SetPenState	$2A	PenStatePtr (L)	Ptr to pen state record
SetPort	$1B	PortPtr (L)	Ptr to new GrafPort
SetPortRect	$1F	Rectangle (L)	Ptr to new port rectangle
SetSCB	$12	ScanLine (W)	Scan line number (0-199)
		NewSCB (W)	New SCB value
SetSolidBackPat	$38	ColorNum (W)	Color # of background pattern
SetSolidPenPat	$37	ColorNum (W)	Color # for pen pattern
SetTextFace	$9A	TextFace (W)	New textface word
SetTextMode	$9C	TextMode (W)	New text drawing mode
SolidPattern	$39	ColorNum (W)	Color number
		PatternPtr (L)	Pointer to pattern for color
StringWidth	$A9	result (W)	Width of string in pixels
		TheString (L)	Ptr to text string

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
TextWidth	$AB	result (W)	Width of text in pixels
		TheText (L)	Ptr to start of text
		Count (W)	Number of characters

Table R6–2: QuickDraw II Error Codes

$0401	QuickDraw II has already been initialized.
$0403	QuickDraw II has not been initialized.
$0410	Screen memory has been reserved.
$0411	The rectangle specified is invalid.
$0420	The pixel chunkiness is not equal.
$0430	The region is already open.
$0431	The region is not open.
$0432	The region has overflowed.
$0433	The region is full.
$0440	The polygon is already open.
$0441	The polygon is not open.
$0442	The polygon is too big.
$0450	Bad color table number.
$0451	Bad color number.
$0452	Bad scan line number.

Table R6–3: The Major Functions in the Window Manager Tool Set ($0E)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
BeginUpdate	$1E	TheWindow (L)	Ptr to window record
CloseWindow	$0B	TheWindow (L)	Ptr to window record
DragWindow	$1A	Grid (W)	Drag resolution (0=default)
		StartX (W)	Starting X coord (global)
		StartY (W)	Starting Y coord (global)
		Grace (W)	Grace distance around bounds
		BoundsRect (L)	Ptr to cursor boundary rect
		TheWindow (L)	Ptr to window record

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
EndUpdate	$1F	TheWindow (L)	Ptr to window record
FindWindow	$17	result (W)	Location code for mouse-down
		WhichWindow (L)	Ptr to space for window ptr
		PointX (W)	X coord to check (global)
		PointY (W)	Y coord to check (global)
FrontWindow	$15	result (L)	Ptr to window record
GetContentDraw	$48	result (L)	Ptr to update subroutine
		TheWindow (L)	Ptr to window record
GetContentOrigin	$3E	result (W)	X coordinate of origin
		result (W)	Y coordinate of origin
		TheWindow (L)	Ptr to window record
GetDataSize	$40	result (W)	Data width
		result (W)	Data height
		TheWindow (L)	Ptr to window record
GetDefProc	$31	result (L)	Ptr to window def. subr.
		TheWindow (L)	Ptr to window record
GetFrameColor	$10	result (L)	Ptr to color table
		TheWindow (L)	Ptr to window record
GetFullRect	$37	result (L)	
		TheWindow (L)	Ptr to window record
GetInfoDraw	$4A	result (L)	Ptr to infobar drawing subr.
		TheWindow (L)	Ptr to window record
GetInfoRefCon	$35	result (L)	RefCon for infobar drawing
		TheWindow (L)	Ptr to window record
GetMaxGrow	$42	result (W)	Maximum window width
		result (W)	Maximum window height
		TheWindow (L)	Ptr to window record

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
GetPage	$46	result (W)	Horizontal distance (page)
		result (W)	Vertical distance (page)
		TheWindow (L)	Ptr to window record
GetScroll	$44	result (W)	Horizontal distance (line)
		result (W)	Vertical distance (line)
		TheWindow (L)	Ptr to window record
GetWFrame	$2C	result (W)	Window frame bit vector
		TheWindow (L)	Ptr to window record
GetWRefCon	$29	result (L)	Reference constant
		TheWindow (L)	Ptr to window record
GetWTitle	$0E	result (L)	Ptr to title string
		TheWindow (L)	Ptr to window record
GrowWindow	$1B	result (W)	New height
		result (W)	New width
		MinWidth (W)	Minimum width of content
		MaxWidth (W)	Maximum width of content
		StartX (W)	Starting X coord (global)
		StartY (W)	Starting Y coord (global)
		TheWindow (L)	Ptr to window record
HideWindow	$12	TheWindow (L)	Ptr to window record
MoveWindow	$19	NewX (W)	X coord of upper-left corner
		NewY (W)	Y coord of upper-left corner
		TheWindow (L)	Ptr to window record
NewWindow	$09	result (L)	Ptr to window record
		ParamList (L)	Ptr to window parameter table
RefreshDesktop	$39	ClobRect (L)	Ptr to redraw rectangle
SelectWindow	$11	TheWindow (L)	Ptr to window record

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
SetContentDraw	$49	ContDraw (L)	Window update procedure
		TheWindow (L)	Ptr to window record
SetContentOrigin	$3F	xOrigin (W)	X coordinate of origin
		yOrigin (W)	Y coordinate of origin
		TheWindow (L)	Ptr to window record
SetDataSize	$41	dataWidth (W)	Data width
		dataHeight (W)	Data height
		TheWindow (L)	Ptr to window record
SetDefProc	$32	DefProc (L)	Window definition procedure
		TheWindow (L)	Ptr to window record
SetFrameColor	$0F	FrColorTbl (L)	Ptr to new color table
		TheWindow (L)	Ptr to window record
SetFullRect	$38	FullRect (L)	Ptr to zoom rectangle
		TheWindow (L)	Ptr to window record
SetInfoDraw	$16	InfoDraw (L)	Information bar procedure
		TheWindow (L)	Ptr to window record
SetInfoRefCon	$36	InfoRefCon (L)	Infobar reference constant
		TheWindow (L)	Ptr to window record
SetMaxGrow	$43	maxWidth (W)	Maximum window width
		maxHeight (W)	Maximum window height
		TheWindow (L)	Ptr to window record
SetPage	$47	hPage (W)	Horizontal distance (page)
		vPage (W)	Vertical distance (page)
		TheWindow (L)	Ptr to window record
SetScroll	$45	hScroll (W)	Horizontal distance (line)
		vScroll (W)	Vertical distance (line)
		TheWindow (L)	Ptr to window record

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
SetSysWindow	$4B	TheWindow (L)	Ptr to system window
SetWFrame	$2D	WFrame (W)	Window frame bit vector
		TheWindow (L)	Ptr to window record
SetWRefCon	$28	WRefCon (L)	New reference constant
		TheWindow (L)	Ptr to window record
SetWTitle	$0D	TitlePtr (L)	Ptr to new title string
		TheWindow (L)	Ptr to window record
ShowWindow	$13	TheWindow (L)	Ptr to window record
SizeWindow	$1C	NewWidth (W)	New width of window
		NewHeight (W)	New height of window
		TheWindow (L)	Ptr to window record
StartDrawing	$4D	TheWindow (L)	Ptr to window record
TaskMaster	$1D	result (W)	Event code
		EventMask (W)	Event mask
		TaskRecord (L)	Ptr to task record
TrackGoAway	$18	result (W)	Boolean: was goaway selected?
		StartX (W)	X coordinate (global)
		StartY (W)	Y coordinate (global)
		TheWindow (L)	Ptr to window record
TrackZoom	$26	result (W)	Boolean: was zoom selected?
		StartX (W)	X coordinate (global)
		StartY (W)	Y coordinate (global)
		TheWindow (L)	Ptr to window record

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
WindShutDown	$03	[no parameters]	
WindStartup	$02	UserID (W)	ID tag for memory allocation
ZoomWindow	$27	TheWindow (L)	Ptr to window record

Table R6–4: Window Manager Error Codes

$0E01	The first word in the NewWindow parameter list is not the correct table size.
$0E02	The window record could not be allocated.
$0E03	Bits 14-31 in the TaskMask field of the task record are not zero.

Table R6–5: Useful Functions in the Font Manager Tool Set ($1B)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
ChooseFont	$16	result (L)	ID for selected font
		currentID (L)	ID of current font
		famSpecs (W)	Family specification
InstallFont	$0E	desiredID (L)	Desired font ID
		scaleword (W)	Scaling factor
FMShutdown	$03	[no parameters]	
FMStartup	$02	SysFontName (L)	Ptr to system font name
		UserID (W)	ID tag for memory allocation
		DPAddr (W)	Pointer to 1 page in bank 0

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
LoadSysFont	$13	[no parameters]	
SetPurgeStat	$0F	fontID (L)	ID for font to purge
		purgeStat (W)	Purge status word

Table R6–6: Font Manager Error Codes

$1B01	The Font Manager has already been started up.
$1B03	The Font Manager is not active.
$1B04	The font family was not found.
$1B05	The font was not found.
$1B06	The font is not in memory.
$1B07	The system font cannot be made purgeable.
$1B08	Illegal family number.
$1B09	Illegal size.
$1B0A	Illegal name length.
$1B0B	FixFontMenu has never been called.

Listing 6–1: How to Create and Display a Window

```
DefineWind START


           PHA                        ;Space for result
           PHA
           PushPtr  TheWindow         ;Pointer to window record
           _NewWindow
           PopLong WindowPtr          ;Save pointer to window record
           RTS


; Window parameters:


TheWindow  DC       I2'WindEnd-TheWindow'     ;Size of table
           DC       I'%1101110110100101'      ;window frame type
           DC       I4'TheTitle'              ;Pointer to window title
           DC       I4'0'                     ;refcon
           DC       I'40,4,182,608'           ;zoom rectangle
           DC       I4'0'                     ;color table (0 = default)
           DC       I'0,0'                    ;document offset
           DC       I'1024,960'               ;height,width of data area
           DC       I'0,0'                    ;height,width max window
           DC       I'9,12'                   ;vert, horiz line movement
           DC       I'144,592'                ;vert, horiz page movement
           DC       I4'0'                     ;info bar refcon
           DC       I2'12'                    ;info bar height
           DC       I4'0'                     ;frame defproc (0 = standard
           DC       I4'DoInfoBar'             ;info bar defproc
           DC       I4'WindUpdate'            ;content defproc
           DC       I'40,4,182,608'           ;Content region rectangle
           DC       I4'-1'                    ;At the front
           DC       I4'0'                     ;Storage (use MM)
WindEnd    ANOP


TheTitle   DC       I1'A GS Window'  ;Window title


WindowPtr  DS       4                ;Pointer to window


           END


; TaskMaster calls this routine whenever an update
; event occurs. It is responsible for redrawing the
; contents of the window.
```

```
WindUpdate START

; Make the data bank equal the code bank so you
; can use absolute addressing:

           PHB
           PHK
           PLB                      ;data bank = code bank

; [insert window drawing code here]

           PLB
           RTL                      ;Do not use RTS!!

           END

; TaskMaster calls this routine when it needs to
; draw the interior of the information bar.
;
; See explanation in listing 6-2.

DoInfoBar  START

; [insert drawing code here]

; Remove 12 bytes of input parameters from the stack:

           LDA     2,S              ;Move return address
           STA     14,S             ; up by three long words
           LDA     1,S
           STA     13,S

           CLC                      ;Raise stack pointer by
           TSC                      ; three long words.
           ADC     #12
           TCS
           RTL

           END
```

Listing 6–2: A Procedure for Drawing an Information Bar

```
; This is an information bar drawing procedure. It centers a
; text string pointed to by the infobar refcon in the bar.
;
; TaskMaster calls InfoProc by pushing three long words on the
; stack (pointer to infobar rectangle, the information bar refcon,
; and a pointer to the window record), then performing a JSL.
; On exit, these parameters must be removed by moving the
; return address (and stack pointer) up by 12 bytes.

; Note: top, left, bottom, right are defined in Standard.Asm.

InfoProc   START

; These are the direct page addresses after aligning the new
; direct page with the stack:

Old_DP     EQU     $01                 ;Old direct page
Old_DB     EQU     Old_DP+1            ;Old data bank
ReturnAddr EQU     Old_DB+2            ;JSL return address
IB_WindPtr EQU     ReturnAddr+3        ;Pointer to window
IB_RefCon  EQU     IB_WindPtr+4        ;Reference constant (string pointer)
IB_RectPtr EQU     IB_RefCon+4         ;Pointer to infobar rectangle

           PHD                         ;Save direct page
           PHB                         ;Save data bank register

           PHK                         ;Set data bank = code bank so we
           PLB                         ; can use absolute addressing

           TSC
           TCD                         ;Align new d.p. with stack

           LDY     #left
           LDA     [IB_RectPtr],Y      ;Get left edge
           STA     IB_Temp

           LDY     #right
           LDA     [IB_RectPtr],Y      ;Get right edge
           SEC
           SBC     IB_Temp             ;Calculate width of rectangle
           PHA

           PHA                         ;Room for result
           PushLong IB_RefCon          ;Pointer to text string
           _StringWidth
           PLA                         ;This is the width of the text
           STA     IB_Temp

           PLA                         ;Get width of rectangle

           CMP     IB_Temp             ;Wider than text?
           BCS     GetMargin           ;Yes, so branch

           LDA     #0                  ;Flush with left side
           BRA     GetMargin1
```

```
GetMargin  SEC
           SBC      IB_Temp              ;Calculate remaining space in rect
           LSR      A                    ;Divide by 2 to get left margin
GetMargin1 STA      IB_Temp
           LDY      #left
           LDA      [IB_RectPtr],Y       ;Add margin to left edge position
           ADC      IB_Temp

           PHA                           ;X-coordinate for MoveTo

           LDY      #bottom
           LDA      [IB_RectPtr],Y       ;Get bottom edge
           SEC
           SBC      #2                   ;(room for descenders)
           PHA                           ;Y-coordinate for MoveTo

           _MoveTo                       ;Position the drawing pen
           PushLong IB_RefCon            ;Push refcon pointer
           _DrawString                   ;Draw string pointed to by refcon

           PLB                           ;Restore data bank register
           PLD                           ;Restore direct page

; Remove parameters from stack before leaving. This is done
; by moving the 3-byte return address, and the stack pointer,
; up by the size of the parameters, then ending with RTL.

           LDA      2,S                  ;Move return address
           STA      14,S                 ; up by three long words
           LDA      1,S
           STA      13,S

           CLC                           ;Raise stack pointer by
           TSC                           ; three long words.
           ADC      #12
           TCS

           RTL                           ;(called with JSL)

IB_Temp    DS       2

           END
```

Listing 6–3: How To Define an STR Macro

```
        MACRO
  &lab  STR    &text
  &lab  DC     I1'L:&text'      ;Length byte
        DC     C"&text"         ;ASCII characters
        MEND
```

Listing 6–4: Using StringWidth To Center a Line of Text

```
; Enter this subroutine with a line number in Y.
; The pointer to the text string must be in A (hiqh) and X (low).

left        GEQU    2
right       GEQU    6

CenterIt    START

            STY     LinePos             ;Save vertical

;Calculate width of string:

            PHA                         ;space for result
            PHA                         ;push pointer (high)
            PHX                         ;push pointer (low)
            _StringWidth
            PopWord TextWidth

;Calculate width of content region:

            PushPtr PortRect
            _GetPortRect                ;Get content rectangle

            SEC
            LDA     PortRect+right      ;Right side minus
            SBC     PortRect+left       ; the left side.

;Left margin is 1/2*(WindowWidth-TextWidth) from left of PortRect:

            SBC     TextWidth
            LSR     A                   ;Divide by 2

            CLC
            ADC     PortRect+left       ;Add to left side

            PHA                         ;Horizontal position
            PushWord LinePos            ;Vertical position
            _MoveTo

            PushPtr TheText
            _DrawString                 ;Draw the string
            RTS

LinePos     DS      2                   ;Vertical line position
```

```
PortRect   DS      8                ;Content rectangle

TextWidth  DS      2

           END
```

Listing 6–5: A Subroutine for Displaying Text in All Sixteen Colors

```
; Use this subroutine in 320x200 mode so that you see all
; 16 colors. (Do this by setting VidMode to $00 and
; XMaxClamp to 320 in Standard.Asm.)

TextColor  START

           PushWord #3
           LDA     #10
           STA     VertPos
           PHA
           _MoveTo                  ;Position on first line

; Just for fun, draw the text in boldface:

           PushWord #%00000001      ;Set the bold bit
           _SetTextFace

           LDX     #0               ;Start with color #0
ColorText  PHX

           PHX
           _SetForeColor            ;Set foreground color

           PushPtr TheText
           _DrawString              ;Print the text

;Move to next line:

           PushWord #3              ;left side
           CLC
           LDA     VertPos
           ADC     #9               ;(height of line)
           STA     VertPos
           PHA                      ;line number
           _MoveTo

           PLX
           INX
           CPX     #16              ;At last color yet?
           BNE     ColorText        ;No, so branch

           RTS
```

```
VertPos     DS        2

TheText     STR       'The quick brown fox jumped.'

            END
```

Listing 6–6: How To Define and Use a QuickDraw Polygon

```
DefinePoly START

; First move the pen to the desired position:

            PushWord #100                ;horizontal
            PushWord #120                ;vertical
            _MoveTo

; ... then open the polygon record:

            PHA                          ;space for result (handle)
            PHA
            _OpenPoly
            PopLong PolyHndl             ;Pop the polygon handle

; ... then draw the outline of the polygon with Line or LineTo
; (this defines a pentagon):

            PushWord #50
            PushWord #0
            _Line
            PushWord #25
            PushWord #-25
            _Line
            PushWord #-50
            PushWord #-25
            _Line
            PushWord #-50
            PushWord #25
            _Line
            PushWord´#25
            PushWord #25
            _Line

; ... then close the polygon record:

            _ClosePoly
            RTS
```

```
; To display the polygon, call ShowPoly:

ShowPoly    ENTRY

            PushLong PolyHndl
            _PaintPoly                 ;(or Frame, Erase, Invert, Fill)
            RTS

; To permanently dispose of the polygon, call ByePoly:

ByePoly     ENTRY

            PushLong PolyHndl
            _KillPoly
            RTS

PolyHndl    DS      4                  ;Handle to polygon record

            END
```

Listing 6–7: The TextReader Program

```
****************************************
*              TextReader              *
*                                      *
* This program demonstrates how to deal *
* with windows that have scroll bars.   *
****************************************
            LIST    OFF
            SYMBOL  OFF
            ABSADDR ON
            INSTIME ON
            GEN     ON

            KEEP    WIND               ;Object code file
            MCOPY   WIND.MAC           ;Macro file

MyCode      START

; Direct page global equates:

TextHndl    GEQU    $0                 ;LONG
TextPtr     GEQU    $4                 ;LONG

            Using   GlobalData
            Using   StartData

            JSR     DoStartUp
```

```
; Define and display the menu bar:

           PushLong #0
           PushPtr  MenuL2
           _NewMenu
           PushWord #0
           _InsertMenu

           PushLong #0
           PushPtr  MenuL1
           _NewMenu
           PushWord #0
           _InsertMenu

           PushWord #1
           _FixAppleMenu                ;Add DAs to Apple menu

           PHA
           _FixMenuBar                  ;Adjust size of menu
           PLA

           _DrawMenuBar

; Save the current direct page

           TDC
           STA      MyDP

           _InitCursor                  ;Turn on arrow cursor

; Ask the user for the name of the file to open. To do this,
; use the SFGetFile dialog box (see chapter 10):

GetName    PushWord #120                ;x
           PushWord #40                 ;y
           PushPtr  SFPrompt            ;prompt
           PushPtr  FilterProc          ;filter procedure
           PushLong #0                  ;(no file type list)
           PushPtr  ReplyRec            ;reply record
           _SFGetFile

           LDA      good                ;Get the result
           BNE      LoadIt              ;Branch if it was "open"
           JMP      DoShutDown

; Load a text file into memory. This is done using
; ProDOS 16 commands (see chapter 10):

LoadIt     _OPEN OpenParms
           LDA      refnum
           STA      refnum1
           STA     refnum2
           STA     refnum3
           STA     refnum4

           _GETEOF EOFParms

           PushLong #0                 ;space for result
           PushLong FileSize           ;Push size of file
           PushWord MyID
           PushWord #$8000             ;Locked
           PushLong #0                 ;(means nothing here)
           _NewHandle
           PopLong TextHndl            ;Save handle to text area

           LDA     [TextHndl]          ;dereference the handle
           STA     TextPtr
           STA     data_buff
           LDY     #2
           LDA     [TextHndl],Y
           STA     TextPtr+2
           STA     data_buff+2

           LDA     FileSize
           STA     request
           LDA     FileSize+2
           STA     request+2

           _NEWLINE NLParms
           _READ ReadParms             ;Read data in to TextPtr block
           _CLOSE CloseParms

;Calculate size of document (in pixel lines). This is done by
; multiplying the number of Carriage Returns by 9 (the height
; of a line):

           STZ     DocSize
           LDY     #0
GetSize    LDA     [TextPtr],Y         ;Get next character
           AND     #$7F
           CMP     #$0D                ;End of line?
           BNE     SkipInc             ;No, so branch

           CLC
           LDA     DocSize
           ADC     #9                  ;9 pixels per line
           STA     DocSize

SkipInc    INY
           CPY     FileSize            ;At end of file?
           BNE     GetSize             ;No, so branch
```

```
          CLC
          LDA     DocSize
          ADC     #20                ;Provide a small bottom margin
          STA     DocSize

; Clip to max depth of 16K ($4000) because of
; QuickDraw boundary restrictions:

          CMP     #$3FFF-9           ;Larger than max?
          BCC     ShowWind           ;No, so branch

          LDA     #$3FFF-9           ;Clip to max (subtract 9 to avoid
          STA     DocSize            ; adverse effects on last line)
; Define and display the window:

ShowWind  PHA                        ;Space for result
          PHA
          PushPtr  MainWindow        ;Pointer to window record
          _NewWindow
          PopLong WindowPtr          ;Save pointer to window record
; Change the title of the window to the name of the file!

SetTitle  PushPtr Filename
          PushLong WindowPtr
          _SetWTitle

EvtLoop   PHA
          PushWord #$FFFF            ;All events
          PushPtr  EventRec
          _TaskMaster
          PLA                        ;Get result code

          CMP     #wInMenuBar        ;Menu item selected?
          BEQ     DoMenu             ;Yes, so branch

          CMP     #wInGoAway         ;In close box?
          BNE     EvtLoop            ;Ignore everything else

          PushLong TextHndl
          _DisposeHandle             ;Free up text block

          PushLong WindowPtr
          _CloseWindow               ;Delete the window
          BRL     GetName            ;Go get another file
```

```
; Handle menu selections:

DoMenu     LDA     TaskData
           AND     #$00FF           ;Convert to 0 base
           ASL     A                ;x2 to step into table
           TAX
           JMP     (MenuTable,X)    ;Call menu item handler

MenuTable  DC      I'DoAboutI'
           DC      I'DoQuitI'

DoAboutI   JSR     FixMTitle
           BRL     EvtLoop

DoQuitI    JSR     FixMTitle
           JMP     DoShutDown

FixMTitle  PushWord #False          ;Highlighting off
           PushWord TaskData+2      ;Get menu ID
           _HiliteMenu
           RTS

           END

; TaskMaster calls this routine whenever an update
; event occurs. It is responsible for redrawing the
; contents of the window.

WindUpdate START
           Using   GlobalData

; Make the data bank equal the code bank so we don't have
; to use absolute long addressing:

           PHB
           PHK
           PLB                      ;data bank = code bank

; Switch to the direct page the application uses so
; that we can use TextPtr:

           PHD                      ;Save current DP
           LDA     MyDP
           TCD                      ;Switch to our DP

; Get the current value of the PortRect. PortRect is
; the rectangle describing the content region.

           PushPtr PortRect
           _GetPortRect
```

```
; Scan for the first line in the document that will
; appear in the window:

            LDA      #8
            STA      YPosition          ;Coord of line #0
            LDY      #$FFFF
            PHY
            BRA      FS0

FindStart   LDA      [TextPtr],Y        ;Get character
            PHY
            AND      #$7F
            CMP      #$0D               ;End of line?
            BNE      FS1                ;No, so branch

            CLC
            LDA      YPosition
            ADC      #9                 ;Add height of line
            STA      YPosition

FS0         LDA      YPosition          ;Get position in document
            CMP      PortRect+top       ;Past the top of PortRect?
            BCS      FS2                ;Yes, so we're ready to start

FS1         PLY
            INY
            CPY      FileSize           ;At end of file?
            BNE      FindStart          ;No, so branch

            PLD
            PLB
            RTL

FS2         PLY
            INY                         ;Move to 1st character of line
            PHY

            PushWord #2                 ;Move to left edge
            PushWord YPosition
            _MoveTo

            PLY

; Determine the number of bytes in the line
; so that we can draw them all at once with DrawText:

NextLine    STY      StartPos
            STZ      Counter
```

```
ShowLine   LDA      [TextPtr],Y        ;Get character
           AND      #$7F               ;Strip high bit
           CMP      #$0D               ;At end of line?
           BEQ      GotLine            ;Yes, so branch

           INC      Counter

           INY
           CPY      FileSize           ;At end of file?
           BNE      ShowLine           ;No, so branch

           DEY

GotLine    PHY

           LDA      Counter            ;CR by itself
           BEQ      GotLine1           ;Yes, so don't draw anything

           CLC                         ;Push starting position
           LDA      StartPos           ; by adding offset to TextPtr
           ADC      TextPtr
           TAX
           LDA      TextPtr+2
           ADC      #0
           PHA
           PHX
           PushWord Counter            ;Number of bytes to draw
           _DrawText

GotLine1   JSR      CRLF               ;Move to next line

; Check to see if we've gone past the bottom of the window:

           PushPtr PenPos
           _GetPen                     ;Get current position

           PLY                         ;Get buffer position back
           INY
           CPY      FileSize           ;At end?
           BEQ      Exit               ;Yes, so branch

           LDA      PortRect+bottom    ;Get bottom of portRect
           CLC
           ADC      #9                 ;go one line further
           CMP      PenPos+v           ;Pen reached bottom yet?
           BCS      NextLine           ;No, so branch

Exit       PLD                         ;Restore DP
           PLB
           RTL                         ;Do not use RTS!!
```

```
PortRect   DS      8                ;rectangle
StartPos   DS      2
Counter    DS      2

           END

***********************************************
* This is the filter procedure for SFGetFile. *
* It is explained in chapter 10.              *
***********************************************
FilterProc START

           PHD                      ;Save direct page
           TSC
           TCD                      ;Align d.p. with stack

           LDY     #16              ;Offset to file type code
           LDA     [$6],Y
           AND     #$00FF           ;(use low byte only)
           CMP     #$B0             ;SRC file?
           BEQ     FP0              ;Yes, so branch
           CMP     #$04             ;TXT file?
           BEQ     FP0              ;Yes, so branch

           LDA     #1               ;1 = display/not selectable
           BRA     FP1

FP0        LDA     #2               ;2 = display/selectable

FP1        PLD                      ;Restore d.p.
           STA     8,S              ;Save the result

           LDA     2,S              ;Move 3-byte return
           STA     6,S              ; address up by 4 bytes.
           LDA     1,S
           STA     5,S

           TSC                      ;Add 4 to the
           CLC                      ; stack pointer.
           ADC     #4
           TCS
           RTL

           END

* Move cursor to left side of next line:

CRLF       START
           Using   GlobalData
           PushPtr PenPos
           _GetPen                  ;Get current pen position

           PushWord #2              ;left edge
           CLC
           LDA     PenPos+v
           ADC     #9               ;9 is the height of the system font
           PHA                      ;New vertical position
           _MoveTo
           RTS

           END

           COPY    STANDARD.ASM     ;Standard startup/shutdown

GlobalData DATA

; Window parameters:

WindowPtr  DS      4                ;Pointer to window

MyTitle    DC      I1'0'            ;Null name; filled in later
MainWindow DC      I2'WindEnd-MainWindow'    ;Size of table
           DC      I'%1101110110100101'      ;window frame type
           DC      I4'MyTitle'               ;Pointer to window title
           DC      I4'0'                     ;refcon
           DC      I'40,4,182,608'           ;zoom rectangle
           DC      I4'0'                     ;color table (0 = default)
           DC      I'0,0'                    ;document offset
DocSize    DC      I'1024,960'               ;height,width of data area
           DC      I'0,0'                    ;height,width max window
           DC      I'9,12'                   ;vert, horiz line movement
PageJumps  DC      I'144,592'                ;vert, horiz page movement
           DC      I4'0'                     ;info bar refcon
           DC      I2'12'                    ;info bar height
           DC      I4'0'                     ;frame defproc (0 = standard)
           DC      I4'0'                     ;info bar defproc
           DC      I4'WindUpdate'            ;content defproc
           DC      I'40,4,182,608'           ;Content region rectangle
           DC      I4'-1'                    ;At the front
           DC      I4'0'                     ;Storage (use MM)
WindEnd    ANOP

; Menu/item lists:

MenuL1     DC      C'> @\N1X',H'0D'       ;Apple menu
           DC      C'##About this program ...\N256V',H'0D'

MenuL2     DC      C'>  File \N2',H'0D'   ;File menu
           DC      C'##Quit\N257*Qq',H'0D'
           DC      C'.'                 ;End of menu
```

```
MyDP       DS      2                 ;Application's direct page
YPosition  DS      2
PenPos     DS      4                 ;Current pen position

; SFGetFile data:

SFPrompt   STR     'Select a file to view:'

ReplyRec   ANOP
good       DS      2                 ;Non-zero if open pressed
filetype   DS      2                 ;ProDOS file type
auxtype    DS      2                 ;ProDOS auxiliary file type
FileName   DS      16                ;Name of file in prefix 0/
fullpath   DS      129               ;Full pathname

; Data for file I/O operations:

OpenParms  ANOP
refnum     DS      2
           DC      I4'FileName'
           DS      4

EOFParms   ANOP
refnum1    DS      2
FileSize   DS      4

NLParms    ANOP
refnum2    DS      2
           DC      I2'0'             ;disable newline read mode
           DS      2

ReadParms  ANOP
refnum3    DS      2
data_buff  DS      4                 ;Pointer to data area
request    DS      4
           DS      4

CloseParms ANOP
refnum4    DS      2

;Event Record for TaskMaster:

EventRec   ANOP
What       DS      2                 ;Event code
Message    DS      4                 ;Event result
```

```
When        DS      4                   ;Ticks since startup
Where       DS      4                   ;Mouse location (global)
Modifiers   DS      2                   ;Status of modifier keys
TaskData    DS      4                   ;TaskMaster data
TaskMask    DC      I4'$00001FFF'       ;TaskMaster handles all

            END
```

CHAPTER 7

Using Pull-down Menus

One of the difficult decisions a programmer faces when developing an application is how to design the command interface between the user and the application. One school of thought says a user should be forced to memorize command names which must be typed in from the keyboard. The advantage of this common technique is that once the commands are mastered, they can be entered very quickly. The two main disadvantages are that it can take a long time to memorize the commands and that commands are easily forgotten if a program is not used for a while.

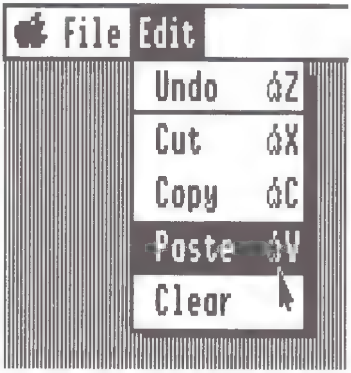
The other extreme is to insist that the user must always select a command from a displayed list of possible choices. Such a list is called a *menu*. This technique is popular with users who are just learning how to use a program, because all the commands are immediately obvious. The disadvantage is it becomes tedious to call up and hunt through a menu once you have mastered the program and know exactly what you want to do.

If you follow Apple's standard user-interface guidelines, you will implement the menu technique with GS applications that use the super high-resolution desktop. To define menus and handle menu activity in standard ways, use the Menu Manager tool set (tool set 15).

Despite its name, the Menu Manager does make concessions to the command-driven interface because you can quickly select certain menu items by pressing a character key while holding down the Open-Apple (Command) key. Such a key is called a *keyboard equivalent*.

The names of each menu defined by a GS application that uses the Menu Manager appear in a rectangular *menu bar* across the top of the graphics screen (see figure 7–1). A user can see the contents of a particular menu by moving the mouse cursor over that menu's title and holding down the mouse button. This causes a rectangle containing the names of all the items in the menu to appear below the menu's title. (Because the effect is like pulling down a window blind, a GS menu is called a *pull-down menu*.) The items are stacked vertically.

Figure 7–1: An Apple IIGS Menu Bar and a Pull-down Menu



Once the menu has been pulled down, a user can select an item by moving the mouse down or up to highlight the appropriate item name and then releasing the mouse button. The user can also inspect an adjacent menu by moving the mouse (with the button still down) to the left or right while it is in the menu bar area.

This chapter covers implementing pull-down menus in GS applications, creating and displaying menu bars, and controlling the appearance of individual items inside menus.

STARTING UP AND SHUTTING DOWN THE MENU MANAGER

Just as with any tool set, the start-up function must be called before the Menu Manager can be used. The function name is MenuStartup. Before calling MenuStartup, however, you must start up QuickDraw, the Event Manager, and the Window Manager—the Menu Manager uses these tool sets to perform some of its functions.

Here is how to start up the Menu Manager:

```
PushWord MyID              ;Program ID (from MMStartup)
PushWord DPAddr            ;Address of direct page area
_MenuStartup
```

DPAddr points to a one-page area in bank zero of memory that the Menu Manager uses for direct page storage. Use NewHandle to allocate this space.

MenuStartup performs all the housekeeping needed to get the Menu Manager up and running. This function creates an empty system menu bar, makes it the current menu bar, and displays it at the top of the screen.

Just before your program ends, call MenuShutDown to free up the memory areas used by the Menu Manager. It requires no parameters and returns no results on the stack.

CREATING A MENU

To create a menu, use the NewMenu function. It requires only one parameter—a pointer to a menu/item line list—and returns a handle to the menu record:

```
    PHA                         ;space for result (long)
    PHA
    PushPtr MenuList            ;Pointer to menu/item list
    _NewMenu
    PopLong MenuHndl            ;Pop handle to the menu
```

Make sure you save the handle returned by NewMenu because you will need it when you add the menu to the menu bar with InsertMenu. Note that if the handle is 0, the menu could not be allocated, either because there is no memory available or because the menu/item line list is invalid.

The menu/item line list is a series of lines, each followed by a null (ASCII $00) or a carriage return (ASCII $0D). A typical list looks something like this, in assembly language format:

```
    TheMenu DC C'>> Edit \N3',I1'0'
            DC C'##Undo\N256V',I1'0'
            DC C'##Cut\N257',I1'0'
            DC C'##Copy\N258',I1'0'
            DC C'##Delete\N259',I1'0'
            DC C'.'
```

The first line in the list defines the title for the menu; the first character (>) informs NewMenu that this is a menu title, not an item name. It is followed by another > character, which simply acts as a place holder for the length of the string. (The Menu Manager fills in the length when you call NewMenu.) The menu title itself comes next, terminated by a backslash and some special characters defining the ID number for the menu. More information about these special characters is given below.

Notice that there is one space to the left and one to the right of the title name. This is not required, but it provides aesthetic gaps between adjacent menu titles in the menu bar.

Subsequent lines define menu items in the order in which they are to appear in the menu. Each line begins with an item character (#) that must be different from the menu title character. Following it are a dummy place holder character, the name of the item, and a terminating backslash followed by special characters.

The list is terminated by a character (.) that is different from the menu item character but could be the same as the menu title character. In fact, if you are defining several menu/item lists in sequence, the menu title character for the next list can be used as a terminator for the current list. Just be sure to follow the last list with a terminator character.

By the way, there is nothing magical in the three characters used in this example. Any characters may be used, as long as the menu item character is different from the menu title character and the terminator character is different from the menu item character.

Following each title and item name in the list is a backslash character (\). The backslash signifies the end of the menu or item name and the beginning of a series of special characters. In the example, the special characters are of the form "Nxxx" where "xxx" represents a decimal ID number for the title or item. Other special characters you can use are summarized in table 7–1. Most of them affect the appearance of the text in the line.

The program in listing 7–1 shows how to create three standard types of menus: an Apple menu, a File menu, and an Edit menu, each using Menu Manager functions.

ID Numbers

Every menu title name must be associated with an ID number from 1 to 65535. None of these numbers are reserved, but each menu title must have a unique number.

The range of ID numbers permitted for item names is narrower. The numbers from 1 to 249 are reserved for use by desk accessory items. The numbers from 250 to 255 are reserved for special editing items and a Close item, which are often needed by desk accessories:

Undo	250	Cancel last editing operation
Cut	251	Cut selected text; put it on the clipboard
Copy	252	Copy selected text to the clipboard
Paste	253	Transfer text from the clipboard to the document
Clear	254	Cut selected text; do not put it on the clipboard
Close	255	Close the active window

(The clipboard is a data area maintained by the Scrap Manager. It facilitates the movement of data within an application or from one application to another.)

The first five items should be placed in a menu called Edit. The Close item should be placed in a menu called File.

Table 7–1: Special Characters for Menu/Item Lists

Special Character	*Meaning*
\	Beginning of special characters
*	Keyboard equivalent characters follow (primary followed by alternate)
B	Boldface the name
C	Mark character follows
D	Disable (dim) the name
H	Two-byte binary ID number follows ($0001 to $FFFF, low-order byte first)
I	Italicize the name
N	Decimal ID number follows (1 to 65535)
U	Underline the name
V	Put a dividing line under the name (but do not use a separate item)
X	Use color-replace highlighting

NOTE: All these special characters may be used with item names. Use only \, D, H, N, and X with menu titles.

When these ID numbers are assigned to these items, and a TaskMaster event loop is used, the special items will automatically be passed to an active desk accessory for processing. See chapter 9 for more information on desk accessories.

The rest of the ID numbers, from 256 to 65535, may be used by the application in any way it sees fit. Keep in mind, however, that each menu item must have a unique ID number, although permanently disabled items may share the same ID number.

When you are defining a series of menus, you might want to number the items consecutively from 256. This makes it possible to access the subroutine to be called when the item is selected by doubling the low-order byte of the item number and using the result as an index into a table of two-byte subroutine addresses. The standard Edit, File, and Desk Accessory items, which have numbers below 256, would be treated as special cases.

Here is a code fragment that illustrates this technique:

```
; Enter here with the menu item number in A:

            CMP #256                ;Standard Edit, Close, or DA?
            BCC DoSpecial           ;Yes, so branch

            AND #$00FF              ;Isolate low-order byte
            ASL A                   ;Double to get index into table
            TAX
            JMP (ITEM_TBL,X)        ;Pass control to item handler

DoSpecial   NOP                     ;Handle special cases here
            RTS

ITEM_TBL    DC I2'DoItem256'        ;Address of item #256 handler
            DC I2'DoItem257'        ;Address of item #257 handler
            DC I2'DoItem258'        ;Address of item #258 handler
```

You can use the H special character to specify the item or menu number in binary, rather than decimal, form. If you use the H, follow it with the two-byte binary number, low-order byte first. For example, to assign an item number of $0103, specify an item line of the form:

```
DC C'##The Item\H',H'03 01',I1'0'
```

You could also use I2'$0103' instead of H'03 01'.

The Appearance of an Item

Several special characters can be used to affect the appearance of an item in a menu. The program in listing 7–2 indicates how to use them.

Three special characters are available for selecting the typeface of an item name: B (boldfaced), I (italicized), and U (underlined). For example, to attach all three faces to an item with an ID of 323, put the following line in the menu/item line list:

```
DC C'##My Item\N323BIU',I1'0'
```

Note, however, that the standard system font used by the GS cannot be underlined. Furthermore, current versions of QuickDraw do not support italics.

To disable an item, use the special character D. A disabled item is dimmed in the menu and cannot be selected when the user pulls down the menu. You should disable an item whenever the action associated with it is not appropriate in the current environment. For example, if you have an item called Open Window, and the window is already open, you should disable the item.

Dividing lines, used to separate groups of related items in a menu, should always be disabled. Here is how to define a disabled dividing line in a menu/item line list:

```
    DC C'##-\N324D',I1'0'
```

The Menu Manager interprets a single hyphen as a row of hyphens extending across the width of the menu.

You can also create a dividing line with the V special character. This type of dividing line is really just an underline and so does not use up the space of an entire item. The main advantage of using it instead of a true dividing line is that you can fit more menu items in a menu.

The X special character denotes a special form of highlighting to be used when a menu title or a menu item is selected. The default highlighting method is called XOR, which causes a name to be inverted. When an X special character is specified, *color-replace* highlighting is used instead of the XOR method; this method causes only the white background of a colored object to be inverted.

You will use color-replace highlighting for a menu whose title is the colored Apple logo. By convention, this menu appears on the left side of a menu bar and contains the names of all the active desk accessories in the system and an "About..." item that displays information about the program. To define the title for the standard Apple menu, use the following line:

```
    DC C'>>@\N1X',I1'0'
```

The Menu Manager substitutes the colored Apple logo for the @ character, as long as there are no additional spaces on either side of the menu title character (@).

Keyboard Equivalents

As you will see later in this chapter, a user may select some menu items by pressing a character key while holding down the Open-Apple modifier key. Use the * special character to assign two keyboard equivalents to a menu item. The two characters immediately following the * are the primary and alternate keyboard equivalents. The primary character is shown to the right of the item name when the menu is pulled down.

If the primary keyboard equivalent is an alphabetic character, it should be in upper case. The alternate equivalent should then be set to the corresponding lowercase character. Here are some examples:

```
    DC C'##Cut\N256*Xx',I1'0'
    DC C'##Help\N303*?/',I1'0'
```

Notice the two standard keyboard equivalents for a help item. The primary character is ? and the alternate is /. Because these two characters are associated with the same key, help will arrive, notwithstanding the status of the Shift key.

Marking Items

The final special character is C. It is used to mark an item by placing a special character to the left of its name in a menu. By convention, an item is to be marked only if a feature associated with it is active, or on. For instance, if you have a menu of font sizes, the one that is currently active should be marked and the others unmarked.

Here is an item line defining an item marked with the character x:

```
DC C'##Underline\N322Cx',I1'0'
```

A more common mark character is the checkmark (ASCII 18), but you cannot put it directly into item lines because it is not a standard keyboard character. Once the menu has been defined, you can set the mark character to a checkmark with the CheckMItem function (see the section below on "Checking and Marking").

CREATING THE MENU BAR

Once you have defined a series of menus with NewMenu, you are ready to add them to the system menu bar. To do this, use the InsertMenu function:

```
PushLong MenuHndl          ;Handle to menu to be inserted
PushWord #0                ;Insert at left side of menu bar
_InsertMenu
```

The second parameter passed to InsertMenu is the ID of the menu after which the menu whose handle is MenuHndl is to be inserted. If the number is 0, as in the example, the menu is inserted before the first menu in the menu bar. The easiest way to add a group of menus to the menu bar is to insert them in order from right to left, using a 0 ID parameter each time.

If your menu bar includes an Apple menu, defined by a >>@\N1X line in the menu/item line list, you should add the names of all the active desk accessories to it, at least if the applications plans to support accessories. Do this using the FixAppleMenu function:

```
PushWord #1                ;Menu ID
_FixAppleMenu
```

The parameter passed to FixAppleMenu is the ID number of the menu to which the names of the desk accessory items are added. The desk accessories are given a consecutive set of ID numbers, beginning with 1. ID numbers from 1 to 249 are reserved for use by desk accessories; when TaskMaster determines that you have selected a desk accessory item, it automatically opens the desk accessory for you. See chapter 9 for more on desk accessories.

Once all the menus have been added to the bar, you must call FixMenuBar to permit the Menu Manager to calculate the height of the menu bar and menus and the maximum width of each menu. The Menu Manager needs this information so that it can draw the menu bar properly. FixMenuBar returns a parameter, the height of the menu bar, so you call it as follows:

```
   PHA                            ;space for result
   _FixMenuBar
   PLA                            ;pop height of menu bar
```

The application probably will not need to know the height of the menu bar, so you can discard the result. You must call FixMenuBar every time you change an item name or menu title so that the Menu Manager can adjust its internal record of the menu dimensions accordingly.

You are now ready to display the menu bar on the screen. For this, use DrawMenuBar—it requires no parameters.

The program in listing 7–1 above illustrates how to create and display a menu bar.

Changing the Name of a Menu

If you wish to change the name of a menu after it has been defined, use SetMenuTitle:

```
            PushPtr NewTitle         ;pointer to name string
            PushWord #258            ;Menu ID code
            _SetMenuTitle

   NewTitle STR 'ANewTitle'
```

NewTitle is a string that is preceded by a length byte. Call DrawMenuBar to redraw the menu bar after changing the name of a title.

CHANGING ITEM ATTRIBUTES

An explanation of how to use special characters in the menu/item line list to set the initial appearance of a menu item was given earlier in this chapter. You can also change an item's appearance, or its name, after the menu has been created, using several Menu Manager functions. Those functions are the subject of this section.

Changing the Name

To change the name of an item, use SetMItem or SetMItemName. SetMItem requires two parameters, a pointer to the new item line and the ID of the item to be renamed:

```
          PushPtr  NewLine          ;Pointer to new item line
          PushWord #256             ;item ID
          _SetMItem
          RTS

 NewLine DC C'##A Different Name',I1'0'
```

The first two characters in the string for the new name (## in the example) are always ignored, as are any characters following a \ character, and the \ character itself. Thus, the new item name will have the same special attributes as the one it is replacing.

You can also use SetMItemName to change the name of an item:

```
          PushPtr  NewName            ;Pointer to new string
          PushWord #256               ;item ID
          _SetMItemName
          RTS

 NewName STR 'A Different Name'
```

In this case, the first parameter is a pointer to a standard string. All other item attributes, such as text style and keyboard equivalents, remain unchanged.

When you change the name of an item in a menu, the width of the menu may change. As a result, you must call CalcMenuSize to permit the Menu Manager to recalculate the width:

```
 PushWord #0      ;0 means: calculate default width
 PushWord #0      ;0 means: calculate default height
 PushWord #256    ;this is the menu ID
 _CalcMenuSize
```

If you do not call CalcMenuSize, and the new item name is too long, some of it will "spill off" into the background when you pull down the menu.

Enabling and Disabling

As mentioned earlier in this chapter, an item can be disabled or enabled. A disabled item is one that is not selectable; it appears dimmed in the menu. An enabled item can be selected and appears in normal script.

You should disable items that are not relevant to the activity currently in progress. If you do so, the user cannot waste time selecting a meaningless activity to perform.

Here is how to disable an item:

```
 PushWord #266             ;Item ID number
 _DisableMItem
```

The corresponding enable function works similarly:

```
PushWord #266              ;Item ID number
_EnableMItem
```

Inserting and Deleting

To insert an item into an already-defined menu, use InsertMItem. It requires three parameters: a pointer to an item line defining the menu item, the ID of the item after which the item is to be inserted, and the ID of the menu to which the item is to be added. Here is how to call InsertMItem:

```
             PushPtr  ItemEntry       ;Pointer to item line
             PushWord #257            ;Item number to add after
             PushWord #3              ;Menu number to add to
             _InsertMItem
             RTS

ItemEntry DC C'##Inserted Item\N333',I1'0'
```

If you specify an item number of 0, the item will be inserted at the top of the menu. An item number of $FFFF causes the item to be added to the end of the menu. If the menu number is 0, the first menu is selected.

The new item definition pointed to by InsertMItem's first parameter has the same format as an entry in a menu/item line list. It must be followed by a null character (0) or a return character (13).

You can also delete items from a menu. For this, use DeleteMItem:

```
PushWord #343              ;ID of item to be deleted
_DeleteMItem
```

You should not use DeleteMItem as a substitute for DisableMItem.

After you have used the InsertMItem or DeleteMItem functions, the vertical size of the menu necessarily will have changed, so call CalcMenuSize as you would after using SetMItem to change an item name.

Checking and Marking

You can check or uncheck a menu item using the CheckMItem function:

```
True     GEQU     $8000

         PushWord #True             ;True = check it
         PushWord #277              ;item ID number
         _CheckMItem
```

The first parameter pushed on the stack is a Boolean instruction indicating whether a check mark (ASCII code 18) is to appear to the left of the item name (true) or whether it should not appear (false). This example checks item 277 because the Boolean parameter is true. Push a value of 0 if you want to uncheck the item.

You can place any character you like to the left of the item name using the SetMItemMark function. Here is how to use a diamond character as the marking character:

```
    PushWord #$13              ;ASCII code for "diamond"
    PushWord #333              ;item ID number
    _SetMItemMark
```

Here are the codes for the four special icons included in the system font:

17	open-apple icon
18	checkmark icon
19	diamond icon
20	solid-apple icon

If you want to remove the marking character, specify an ASCII code of 0 for the marking character. To determine which character is currently marking an item, use GetMItemMark:

```
    PushWord #0                ;space for result
    PushWord #333              ;item ID number
    _GetMItemMark
    PLA                        ;Result contains character code
```

If the item specified is not marked, the result is 0.

Changing the Text Style

Item names can be drawn in plain text or they can be boldfaced, italicized, underlined, outlined, or shadowed. Select a text style by passing a style word to SetMItemStyle as follows:

```
    PushWord #%00000001        ;style word (bold)
    PushWord #267              ;item ID number
    _SetMItemStyle
```

Five bits in the style word enable the five fundamental style attributes:

bit 0	bold
bit 1	italic
bit 2	underline
bit 3	outline
bit 4	shadow
bit 5–15	zero

To select a particular style, set the appropriate bit to 1. The style types are not mutually exclusive, so you can combine them as you like. Use a style word of 0 if you want the item name drawn in plain text.

- *Note:* Early versions of QuickDraw do not support italic, outline, or shadow. In addition, fonts with a descent of 0 and 1 cannot be underlined; this includes the default system font used on the GS.

To determine the current style of an item, use GetMItemStyle:

```
    PushWord #0             ;space for result
    PushWord #267           ;item ID number
    _GetMItemStyle
    PLA                     ;pop the result (a style word)
```

The result is a style word.

REMOVING MENUS

To remove a menu definition from the system permanently, thus freeing up the memory it uses, first remove it from the system menu bar with DeleteMenu:

```
    PHA                     ;space for result (handle)
    PHA
    PushWord MenuID         ;ID number of menu
    _GetMHandle
    _DeleteMenu
```

Notice that GetMHandle is used here to determine the handle to the system menu. Of course, if you saved the handle returned by NewMenu, you could just call DeleteMenu after pushing the handle on the stack.

Next, call DisposeMenu (this code assumes that you have saved the menu handle at MenuHndl):

```
PushLong MenuHndl        ;Handle to menu
_DisposeMenu
```

Once you have disposed of a menu like this, you cannot use it again unless you redefine it with NewMenu.

To reinstall a menu that was removed with DeleteMenu but that was not disposed of, use the InsertMenu function described earlier in this chapter.

Note that neither DisposeMenu nor DeleteMenu has an immediate effect on the appearance of the menu bar on the screen. To redraw the menu bar without the removed menu, call FixMenuBar to recalculate the menu bar size, then call DrawMenuBar.

USER INTERACTION

A vital part of any program using menus is the code that handles activity in the menu bar area in a manner consistent with Apple's user-interface guidelines. There are two general ways of doing this, depending on whether you are using a TaskMaster event loop or a GetNextEvent event loop.

Using GetNextEvent

Handling menu bar activity if you are using GetNextEvent requires the most work. The program in listing 7–3 shows what to do. When GetNextEvent returns a mouse-down event (code 1), it calls FindWindow to determine if the event occurred in the menu bar area. If it did, FindWindow returns a result of wInMenuBar (17) and the program calls MenuSelect.

MenuSelect tracks the movement of the mouse until the mouse button is released. It manages all the pull-down menu chores, including highlighting appropriate menu titles and item names. It returns a result indicating which menu item was selected, if any.

Here is the calling sequence for MenuSelect:

```
PushPtr TaskRecord     ;Pointer to task record
PushLong #0            ;0 = system menu bar
_MenuSelect
```

The task record used by this call is the GetNextEvent event record that returned the mouse-down event, followed by the long-word TaskData and TaskMask fields. MenuSelect returns its result in the TaskData field: the low-order word is the ID of the menu item selected, and the high-order word is the ID of the menu selected. (TaskMask is actually used by TaskMaster only.) The program can then take whatever action is appropriate for the menu item selected. If the result is 0, no menu item was selected.

The program in listing 7–3 also checks for keyboard equivalents of menu items. As was mentioned earlier in this chapter, an item can be associated with a keyboard equivalent using the * special character. To select such a menu item, you just tap the appropriate key while holding down the Open-Apple key.

To deal with keyboard equivalents, the program checks for key-down and autokey events. When it finds one of these events, it passes the TaskRecord to MenuKey for analysis. MenuKey, like MenuSelect, returns the ID of the menu item selected in the low-order word of the TaskData field. If the keystroke did not correspond to the primary or alternate equivalent, MenuKey returns an ID of 0.

Using TaskMaster

It is much easier to deal with activity in the menu bar using TaskMaster than it is using GetNextEvent, because TaskMaster automatically calls FindWindow and MenuSelect to determine what menu item was selected. It also calls MenuKey to check for keyboard equivalents. All you have to do is make sure that menu-bar handling has been enabled in the TaskMask. The program in listing 7–4 shows what a TaskMaster menu-handling routine looks like.

The event code returned by TaskMaster when a menu item has been selected is wInMenuBar (17). This is the same code returned by FindWindow for a mouse-down event in the menu bar. The ID numbers of the selected menu and menu item are stored in the high- and low-order words of the TaskData field of the Task Record, respectively.

Generally speaking, TaskMaster simply returns the ID number of the menu item selected—it does not attempt to process the selection in any way (that is up to the application). The exceptions involve desk accessory items, which have menu IDs from 1 to 249, and special menu items, which have IDs from 250 to 255.

If a desk accessory item is selected, TaskMaster automatically opens the desk accessory in question and returns a null result. If a desk accessory window is active, special menu items are processed by passing them to the desk accessory for action. The desk accessory handles the Close item (#255) by closing its window. Editing items (#250 to #254) may or may not be handled by the accessory (see chapter 9); if they are not, TaskMaster returns a wInSpecial (25) event code to give the application a chance to do something with it.

Removing Menu Title Highlighting

If the user selects a menu item, both MenuSelect and MenuKey highlight the title of the menu in which it appears. When processing of the commands ends, you should call HiliteMenu to return the title to its normal appearance:

```
    PushWord #0              ;0 = normal title
    PushWord TaskData+2      ;Menu ID number
    _HiliteMenu
```

The first parameter is a Boolean parameter indicating whether the menu title is to be highlighted (true) or drawn normally (false). The value passed here is false (0), so the title is redrawn normally. The second parameter is the ID number of the menu. It is stored in the high-order word of the TaskData field of the event record.

COLOR AND THE MENU MANAGER

By default, menu bars and pull-down menus are white and any text items inside them are black. The menu bar outline, menu outline, underlines, and dividing lines are also black. When items are highlighted, the text becomes white and the background black.

With the SetBarColors function you can change the colors the Menu Manager uses to display menu bars, menus, and the text items in them. The colors used for unselected items, selected items, and outlines can be set separately.

Here is how to call SetBarColors:

```
    PushWord NewBarColor       ;unselected color
    PushWord NewInvertColor    ;selected color
    PushWord NewOutColor       ;outline color
    _SetBarColors
```

Each of the three parameters defines the colors of two areas, as follows:

PARAMETER	BITS 0-3	BITS 4-7
NewBarColor	Text color when item is not selected	Background color when item is not selected
NewInvertColor	Text color when item is selected	Background color when item is selected
NewOutColor	[zero]	Color of outline of menu and menu bar, underlines, and dividing lines

Bits 8–15 must always be 0, unless you specify a negative parameter (bit 15 = 1). When a parameter is negative, the color scheme of a given attribute does not change.

In 640-by-200 mode you can use color numbers from 0 to 3. In 320-by-200 mode, you can use color numbers from 0 to 15. The standard colors assigned to each number were given in table 6–1 in chapter 6.

Table R7–1: The Major Functions in the Menu Manager Tool Set ($0F)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
CalcMenuSize	$1C	NewWidth (W)	Menu width (0=automatic)
		NewHeight (W)	Menu height (0=automatic)
		MenuID (W)	ID of menu
CheckMItem	$32	CheckItem (W)	True = check/False = uncheck
		ItemID (W)	ID of menu item
DeleteMItem	$10	ItemID (W)	ID of menu item to delete
DeleteMenu	$0E	MenuID (W)	ID of menu to delete
DisableMItem	$31	ItemID (W)	ID of menu item to disable
DisposeMenu	$2E	MenuHandle (L)	Handle to menu to dispose
DrawMenuBar	$2A	[no parameters]	
EnableMItem	$30	ItemID (W)	ID of menu item to delete
FixMenuBar	$13	result (W)	Height of menu bar
GetMHandle	$16	result (L)	Handle to menu
		MenuID (W)	ID of menu
GetMItemMark	$34	result (W)	Mark character (0=no mark)
		ItemID (W)	ID of menu item
GetMItemStyle	$36	result (W)	Text style
		ItemID (W)	ID of menu item
HiliteMenu	$2C	HiliteFlag (W)	True=highlight/False=normal
		MenuID (W)	ID of menu

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
InsertMItem	$0F	AddItemPtr (L)	Ptr to item definition to use
		InsertAfter (W)	ID of item to insert after
		MenuID (W)	ID of menu to contain item
InsertMenu	$0D	AddMenuHndl (L)	Handle to menu to insert
		InsertAfter (W)	ID of menu to insert after
MenuKey	$09	TaskRecPtr (L)	Ptr to task record
		MenuBarPtr (L)	Ptr to menu bar (0=system)
MenuSelect	$2B	TaskRecPtr (L)	Ptr to task record
		MenuBarPtr (L)	Ptr to menu bar (0=system
MenuShutDown	$03	[no parameters]	
MenuStartup	$02	UserID (W)	ID tag for memory allocation
		DPageAddr (W)	Address of 1 page in bank 0
NewMenu	$2D	result (L)	Handle to menu record
		MenuList (L)	Ptr to menu list record
SetBarColors	$17	NewBarColor (W)	Normal item color
		NewInvColor (W)	Item color when selected
		NewOutColor (W)	Outline color
SetMItemName	$23	NewItemStr (L)	Ptr to new menu item string
		ItemID (W)	ID of menu item
SetMenuTitle	$21	NewTitleStr (L)	Ptr to new menu title string
		MenuID (W)	ID of menu

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
SetMItem	$24	NewLinePtr (L)	Ptr to new item line
		ItemID (W)	ID of menu item
SetMItemMark	$33	MarkItem (W)	ASCII code for mark character
		ItemID (W)	ID of menu item
SetMItemStyle	$35	TextStyle (W)	New text style
		ItemID (W)	ID of menu item

Listing 7–1: A Subroutine for Defining Menus and a Menu Bar

```
****************************************
* This subroutine shows how to define   *
* standard Apple-File-Edit menus and a  *
* system menu bar.                      *
****************************************
DoMenus     START

; Start defining the individual menus and adding them
; to the menu bar. This is done from right to left order
; for simplicity.

            PushLong #0             ;0 = system menu bar
            PushPtr  MenuL3         ;Pointer to menu definition
            _NewMenu                ;Define the Edit menu
            PushWord #0             ;Add to menu
            _InsertMenu

            PushLong #0
            PushPtr  MenuL2
            _NewMenu                ;Define the File menu
            PushWord #0
            _InsertMenu

            PushLong #0
            PushPtr  MenuL1
            _NewMenu                ;Define the Apple menu
            PushWord #0
            _InsertMenu

            PushWord #1             ;Menu ID (1 = Apple menu)
            _FixAppleMenu           ;Add DAs to Apple menu

            PHA
            _FixMenuBar             ;Set the menu size
            PLA

            _DrawMenuBar            ;Display the menu bar
            RTS

; Menu/item lists:

MenuL1      DC      C'>> @\N1X',H'0D'       ;Apple menu
            DC      C'##About this program ...\N256V',H'0D'

MenuL2      DC      C'>>  File \N2',H'0D'   ;File menu
            DC      C'##Close\N255V',H'0D'  ;Special close item
            DC      C'##Quit\N257*Qq',H'0D'
```

```
MenuL3     DC      C'>  Edit \N3',H'0D'  ;Edit menu
           DC      C'##Undo\N250V*Zz',H'0D'    ;Special edit items
           DC      C'##Cut\N251*Xx',H'0D'
           DC      C'##Copy\N252*Cc',H'0D'
           DC      C'##Paste\N253*Vv',H'0D'
           DC      C'##Clear\N254',H'0D'

           DC      C'.'                ;End of menu

           END
```

Listing 7–2: Changing the Appearance of Items in a Menu Using Special Characters

```
MenuL4     DC      C'>> Attributes \N4',H'0D'
           DC      C'##Bold\N258B',H'0D'
           DC      C'##Italic\N259I',H'0D'
           DC      C'##Underline\N260U',H'0D'
           DC      C'##All Attributes\N261BIU',H'0D'
           DC      C'##Disabled\N262D',H'0D'
           DC      C'##-\N263D',H'0D'
           DC      C'##Marked\N264C',H'13',H'0D'
           DC      C'##Checked\N265',H'0D'
           DC      C'##Equivalent B\N266*Bb',H'0D'
           DC      C'##Dividing\N267V',H'0D'
           DC      C'##Normal Item\N268',H'0D'
           DC      C'.'                      ;End of menu
```

Listing 7–3: Handling Menu Activity When Using a GetNextEvent Event Loop

```
wInMenuBar GEQU    17                  ;In menu bar

MenuDemo   START

EvtLoop    PHA
           PushWord #$FFFF             ;All events
           PushPtr  TaskRec
           _GetNextEvent
           PLA                         ;Did anything happen?
           BEQ     EvtLoop             ;No, so branch

           LDA     what                ;Get event code
           CMP     #1                  ;Mouse-down?
           BEQ     FindWhere           ;Yes, so branch

           CMP     #3                  ;Key-down?
           BEQ     EquivChk            ;Yes, so check for equivalent
           CMP     #5                  ;Auto-key?
           BEQ     EquivChk            ;Yes, so branch
```

```
DoOther     ANOP

; [handle other types of events here]

            BRL     EvtLoop

;Check for keyboard equivalents of menu items:

EquivChk    PushPtr TaskRec
            PushLong #0                 ;0 = system menu bar
            _MenuKey
            LDA     TaskData            ;menu item selected?
            BNE     DoMenu              ;Yes, so handle it

            LDA     what                ;Get code back
            BRL     DoOther

;Find out where the mouse-down took place:

FindWhere   PHA                         ;Space for result
            PushPtr TheWindow           ;return window pointer here
            PushLong where              ;push point to check (global)
            _FindWindow
            PLA                         ;Get result code

            CMP     #wInMenuBar         ;In menu bar?
            BEQ     WasMenu             ;Yes, so branch

; [handle mouse activity in other areas here]

            BRL     EvtLoop

WasMenu     PushPtr TaskRec
            PushLong #0                 ;0 = system menu bar
            _MenuSelect                 ;Determine which menu item

; Handle menu selections:

DoMenu      LDA     TaskData            ;Get menu item ID
            BEQ     EvtLoop             ;Branch if nothing selected

            CMP     #256                ;Is it a special item (ID < 256)?
            BCC     DoMenu1             ;Yes, so branch

; This code assumes that standard menu items are numbered
; consecutively from 256.
```

```
           AND     #$00FF          ;Convert to 0 base
           ASL     A               ;x2 to step into table
           TAX
           JSR     (MenuTable,X)   ;Call item subroutine
           BRA     TitleOff

DoMenu1    JSR     DoSpecialI

TitleOff   PushWord #0             ;0 = title highlighting off
           PushWord TaskData+2     ;The menu ID is at TaskData+2
           _HiliteMenu

           BRL     EvtLoop

; Table of subroutine addresses, in numeric order:

MenuTable  DC      I'DoItem256'
           DC      I'DoItem257'
           DC      I'DoItem258'

; Handle special editing items, close item here:
; Normally would pass these to the Desk Manager
; with SystemEdit.

DoSpecialI ANOP

           CMP     #255            ;Close item?
           BNE     DoEdit          ;No, so it must be Edit item

; Put code here to close the active window.

           RTS

; Here is where to put code to handle the special edit items.
; The ID codes are 250 (Undo), 251 (Cut), 252 (Copy),
; 253 (Paste), and 254 (Clear).

DoEdit     SEC
           SBC     #250            ;Convert to 0 base
           ASL     A               ;x2 to step into table
           TAX
           JMP     (EditTable,X)

EditTable  DC      I'DoUndo'
           DC      I'DoCut'
           DC      I'DoCopy'
           DC      I'DoPaste'
           DC      I'DoClear'
```

```
DoUndo      ANOP
DoCut       ANOP
DoCopy      ANOP
DoPaste     ANOP
DoClear     ANOP
            RTS

TheWindow   DS      4                   ;Returned by FindWindow

;GetNextEvent task record:

TaskRec     ANOP
What        DS      2                   ;Event code
Message     DS      4                   ;Event result
When        DS      4                   ;Ticks since startup
Where       DS      4                   ;Mouse location (global)
Modifiers   DS      2                   ;Status of modifier keys
TaskData    DS      4                   ;TaskMaster data
TaskMask    DC      I4'$00001FFF'       ;TaskMaster handles all

            END
```

Listing 7–4: Handling Menu Activity When Using a TaskMaster Event Loop

```
wInMenuBar GEQU     17                  ;In menu bar
wInSpecial GEQU     25                  ;In special menu item

MenuDemo    START

EvtLoop     PHA
            PushWord #$FFFF             ;All events
            PushPtr  TaskRec
            _TaskMaster
            PLA                         ;Get result code

            CMP     #wInMenuBar         ;Menu item selected?
            BEQ     DoMenu              ;Yes, so branch
            CMP     #wInSpecial         ;Special menu item?
            BEQ     DoMenu              ;Yes, so branch

; handle other result codes here

            BRL     EvtLoop

; Handle menu selections:

DoMenu      LDA     TaskData

            CMP     #256                ;Is it a special item (ID < 256)?
            BCC     DoMenu1             ;Yes, so branch
```

```
; This code assumes that standard menu items are numbered
; consecutively from 256.

            AND      #$00FF             ;Convert to 0 base
            ASL      A                  ;x2 to step into table
            TAX
            JSR      (MenuTable;X)      ;Call item subroutine
            BRA      TitleOff

DoMenu1     JSR      DoSpecialI

TitleOff    PushWord #0                 ;0 = title highlighting off
            PushWord TaskData+2         ;The menu ID is at TaskData+2
            _HiliteMenu

            BRL      EvtLoop

; Table of subroutine addresses, in numeric order:

MenuTable   DC       I'DoItem256'
            DC       I'DoItem257'
            DC       I'DoItem258'

; Handle special editing items, close item here:

DoSpecialI ANOP

            CMP      #255               ;Close item?
            BNE      DoEdit             ;No, so it must be Edit item

; Put code here to close the active window.

            RTS

; Here is where to put code to handle the special edit items.
; The ID codes are 250 (Undo), 251 (Cut), 252 (Copy),
; 253 (Paste), and 254 (Clear).

DoEdit      SEC
            SBC      #250               ;Convert to 0 base
            ASL      A                  ;x2 to step into table
            TAX
            JMP      (EditTable,X)

EditTable   DC       I'DoUndo'
            DC       I'DoCut'
            DC       I'DoCopy'
            DC       I'DoPaste'
            DC       I'DoClear'
```

```
DoUndo      ANOP
DoCut       ANOP
DoCopy      ANOP
DoPaste     ANOP
DoClear     ANOP
            RTS

;TaskMaster task record:

TaskRec     ANOP
What        DS      2                    ;Event code
Message     DS      4                    ;Event result
When        DS      4                    ;Ticks since startup
Where       DS      4                    ;Mouse location (global)
Modifiers   DS      2                    ;Status of modifier keys
TaskData    DS      4                    ;TaskMaster data
TaskMask    DC      I4'$00001FFF'        ;TaskMaster handles all

            END
```

CHAPTER 8

Using Dialog and Alert Boxes

Apple's standard user-interface guidelines describe two special types of windows called *dialog boxes* and *alert boxes*. The general appearance of these types of windows is shown in figures 8–1 and 8–2. Dialog boxes are conventionally used to request certain types of input from the user. They can include several data input fields, containing such items as a line of text that can be edited, check boxes, buttons, and scroll controls. They can also contain static items which cannot be modified, such as text strings, icons, and pictures.

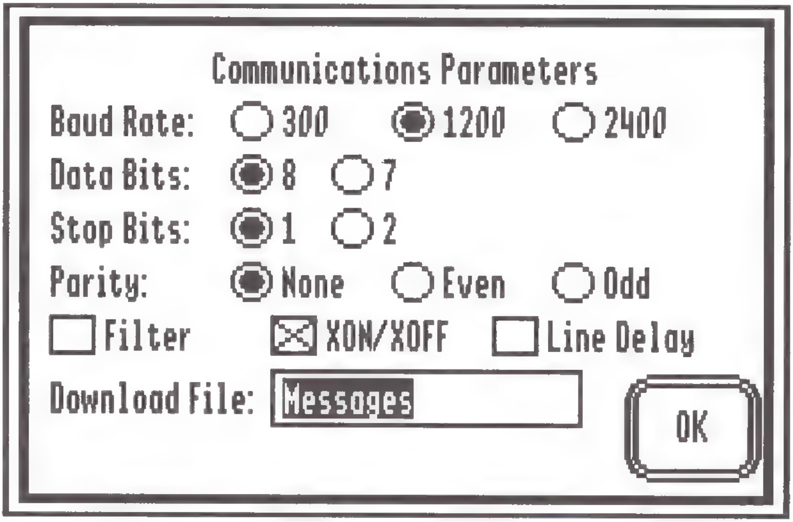
An alert box, as its name suggests, normally warns a user of the consequences of a proposed action which might result in the destruction or loss of data. In a typical application, an alert box contains "OK" and "Cancel" buttons that can be clicked to either verify the action or abort it. For instance, if you are running a disk utility program and you try to format a disk, you will invariably see an alert box warning you that the operation will destroy data and asking you to verify that you wish to proceed.

Alert boxes can be used to display status information as well. Most "About..." items in the standard Apple menu use alert boxes to display authorship and copyright information, for example.

The main difference between alerts and dialogs is that alerts do not contain any user-alterable input areas, such as a text editing box. They contain only static items and one or more buttons you can click to dismiss the alert and continue with the main application.

To create and control dialog and alert boxes, you must use a tool set called the Dialog Manager (tool set 21). This chapter investigates the Dialog Manager and explains how you can use it in your applications.

Figure 8–1. A Dialog Box



Communications Parameters

Baud Rate: ○ 300 ● 1200 ○ 2400

Data Bits: ● 8 ○ 7

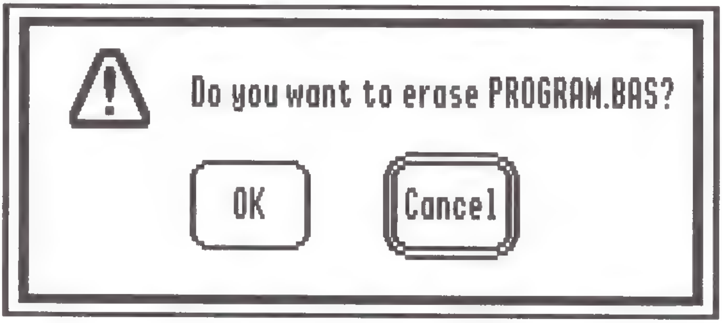
Stop Bits: ● 1 ○ 2

Parity: ● None ○ Even ○ Odd

□ Filter ☒ XON/XOFF □ Line Delay

Download File: Messages

Figure 8–2. An Alert Box



Do you want to erase PROGRAM.BAS?

STARTING UP THE DIALOG MANAGER

To start up the Dialog Manager, pass the ID of the program (the one returned by MMStartup) to DialogStartup:

```
    PushWord MyID              ;Program ID
    _DialogStartup
```

Calling DialogStartup prepares the Dialog Manager for activity and initializes all internal variables and routines.

Because the Dialog Manager uses QuickDraw II, the Window Manager, the Menu Manager, the Control Manager, and the LineEdit tool sets, you must start them up before calling DialogStartup. The STANDARD.ASM program in chapter 3 takes care of this for you.

The only tool set in this list that has not been discussed before is LineEdit, a tool set used for editing lines of text. To start it up, you must provide one page in bank $00 that it can use as a direct page. Here is the start-up sequence:

```
    PushWord MyID            ;ProgramID
    PushWord DPSpace         ;Address of one page DP area
    _LEStartup
```

The standard LineEdit editing commands will be examined later in this chapter.

When you have finished using the Dialog Manager, shut it down with the DialogShutDown function. This function expects no input parameters and returns no results. You can shut down LineEdit by calling LEShutDown in the same way.

CREATING DIALOG BOXES

Two general classes of dialog boxes can be implemented on the GS: modal and modeless. A modal dialog box is one that, once displayed, handles all keyboard and mouse events until the user dismisses the box by clicking a button in the box. Mouse clicks outside the dialog box are ignored, so you cannot pull down a menu, select another window, or use a desk accessory until the modal dialog box is dismissed. In fact, this is how the modal dialog box gets its name: when you are using it, you are confined to a special operating mode until a button is clicked.

A modeless dialog box, on the other hand, is just like any other window on the screen, and TaskMaster treats it as such. It has a goaway box and a title bar, but no zoom box, grow box, or scroll controls. The user is free to switch between the modeless dialog box and any other window on the screen in the usual way. You can remove a modeless dialog box from the screen just as you would any other window: by clicking its goaway box or selecting the Close item from a File menu when the window is active.

Modal and modeless dialog boxes are defined and created using the same general programming techniques. The difference in their behavior arises because different instructions are used to interact with them while they are on the screen.

Modal Dialog Boxes

To create a modal dialog box, you can use either NewModalDialog or GetNewModalDialog. The main difference between the two is that all the parameters for a NewModalDialog call are pushed on the stack. With GetNewModalDialog, the only parameter is a pointer to a dialog template; the template contains all the information needed to display the dialog box correctly.

You will probably prefer to use GetNewModalDialog, because it requires less work. Here is how to call it:

```
    PHA                          ;Space for result (long)
    PHA
    PushPtr DlogTemp             ;Pointer to dialog template
    _GetNewModalDialog
    PopLong DialogPtr            ;Pop dialog pointer
```

An example of how to use GetNewModalDialog is shown in listing 8–1. It creates the dialog box shown in figure 8–1.

GetNewModalDialog draws the dialog box on the screen, but does not actually draw the items inside the box (that is done by ModalDialog; see below). The result returned by GetNewModalDialog is a pointer to the internal record that the Dialog Manager maintains in order to keep track of the modal dialog box. Save it for use with other dialog-related functions.

The dialog template required by GetNewModalDialog is a variable length table containing information about the size of the dialog box and the items it is to contain. Here is its structure:

```
    DlogTemp    ANOP
                DC      I't,l,b,r' ;Content rectangle (global)
                DC      I'$FFFF'   ;Non-zero = Visible
                DC      I4'0'      ;Reference constant
                DC      I4'Item1'  ;Pointer to 1st item template
                DC      I4'Item2'  ;Pointer to 2nd item template
                ...
                DC      I4'ItemN'  ;Pointer to Nth item template
                DC      I4'0'      ;Long zero terminator
```

The template begins with the rectangular dimensions of the content portion of the dialog box, in global coordinates. These dimensions are followed by a Boolean value that indicates whether or not the dialog box is to be visible; you should make this value non-zero (true) so that the box will appear on the screen. The third parameter is a reference constant that the application can use for anything it likes.

Next comes a series of pointers to the templates for the various items to be drawn inside the dialog box. These item templates will be discussed in the next section. The last item pointer is followed by a long zero, which marks the end of the list.

If you choose to use NewModalDialog, you must provide three input parameters: a pointer to the content rectangle, the visible flag, and the reference constant:

```
                PHA                          ;Space for result
                PHA
                PushPtr BoundsRect           ;Pointer to rectangle def
```

```
            PushWord #$8000         ;Visible Boolean
            PushLong #0             ;Reference constant
            _NewModalDialog
            PopLong DialogPtr       ;Pop dialog pointer
            RTS

    BoundsRect DC I't,l,b,r'        ;Rectangle definition
```

Notice that the item definitions are missing. After calling NewModalDialog, you must attach items to the dialog record using the NewDItem or GetNewDItem functions. Both are described later in this chapter.

ITEM TYPES

As mentioned above, the dialog template you pass to GetNewModalDialog contains pointers to the definition templates for each item that is to appear in the dialog box. The items supported by the Dialog Manager are as follows:

- Button
- Check box
- Radio button
- Scroll bar
- User-defined control (two types)
- Static text (up to 255 characters)
- Long static text (up to 32,767 characters)
- Editable line
- Icon
- QuickDraw II picture
- User-defined item

The standard symbolic names for these items are shown in table 8–1. Figure 8–3 shows what the standard item types look like.

The template for any item has the following structure:

```
    ItemTemp ANOP
             DC       I'ItemID'        ;Unique ID code for the item
             DC       I4't,l,b,r'      ;The item rectangle
             DC       I'ItemType'      ;Item type code
             DC       I4'ItemDescr'    ;Item descriptor (item specific)
             DC       I'ItemValue'     ;Initial value of item
```

Table 8–1: Item Type Codes for Dialog and Alert Boxes

Symbolic Name	*Code*	*Description*
ButtonItem	$000A	Button control
CheckItem	$000B	Check box control
RadioItem	$000C	Radio button item
ScrollBarItem	$000D	Scroll bar control
UserCtlItem	$000E	User-defined control
StatText	$000F	Static text
LongStatText	$0010	Long static text
EditLine	$0011	Editable line
IconItem	$0012	Icon
PicItem	$0013	QuickDraw II picture
UserItem	$0014	User-defined item
UserCtlItem2	$0015	User-defined control
ItemDisable	$8000	Add this to disable

NOTE: Add the constant ItemDisable to the code for an item to disable that item.

```
         DC      I'ItemFlag'      ;Display flags (item specific)
         DC      I4'ItemColor'    ;Pointer to item's color table
```

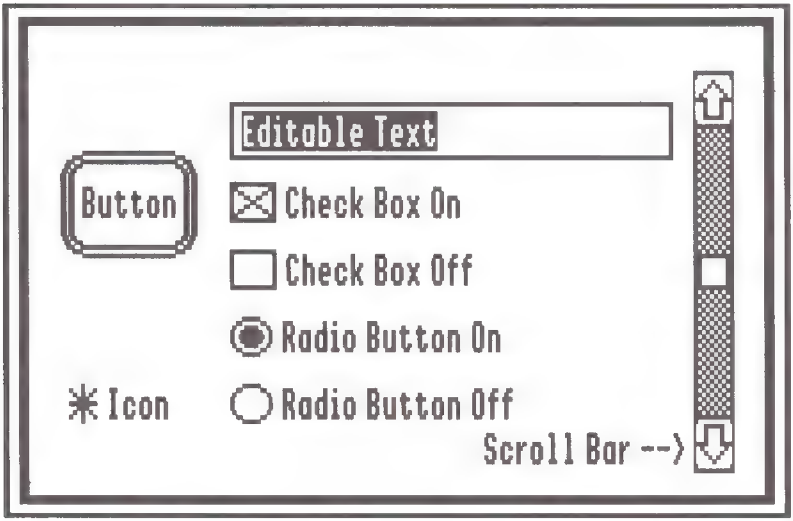
The ItemID is an identification number you assign to the item. No other item in the dialog box may use the same number.

ItemRect is the rectangle in which the item will be drawn. The coordinates are in standard top, left, bottom, right order. You will find that the most frustrating aspect of defining an item is adjusting its rectangle so that it appears in the correct position and does not overlap other items.

ItemType is the code the Dialog Manager uses to identify the type of control you are defining. The values for each type of item are shown in table 8–2. The symbolic names for these codes are defined in the STANDARD.ASM file with GEQU directives.

The meaning of ItemDescr is different for each item type. It is generally a pointer to a data area associated with the item. For a static text item, for example, it points to the text string.

Figure 8–3. Common Items Used in Dialog Boxes



ItemValue contains the initial value of the item. As will be discussed, control items like check boxes, radio buttons, and scroll bars have values associated with them.

The primary purpose of ItemFlag is to indicate whether the item is to be visible or invisible. For visible items, bit 7 must be 0; for invisible items, bit 7 must be 1. Certain other items, namely buttons and scroll bars, use other bits in ItemFlag to control their appearance. Radio buttons use six bits as a family number so that related radio buttons can be dealt with as a group.

ItemColor points to a color table for the item. Set it to 0 to use the standard color table.

Each of the item types supported by the Dialog Manager is described below.

Buttons

A button item (ButtonItem) is a rectangle that the user can click to dismiss the dialog box. The application should react to the click by saving the values of any other items in the dialog box, removing the box from the screen, and then taking whatever action is appropriate for the button selected.

The ItemDescr field in an item template points to the name of the button (preceded by a length byte); the name is drawn inside the button rectangle. ItemValue is not used.

Table 8–2: The Contents of the ItemDescr, ItemValue, and ItemFlag Fields in an Item Template

Item Type	*ItemDescr*	*ItemValue*
ButtonItem	Pointer to item name	[Not used]
CheckItem	Pointer to item name	0=off, 1=on
RadioItem	Pointer to item name	0=off, 1=on
ScrollBarItem	Pointer to action procedure	Initial value: 0 to 290
UserCtlItem	Pointer to control procedure	Initial value
StatText	Pointer to static string	[Not used]
LongStatText	Pointer to text	Text length: 0 to 32767
EditLine	Pointer to default string	Maximum length: 0 to 255
IconItem	Handle to icon definition	[Not used]
PicItem	Handle to picture	[Not used]
UserItem	Pointer to definition procedure	[Not used]
UserCtlItem2	Pointer to control parameters	Initial value

Item Type	*ItemFlag*
ButtonItem	Bit 0 : 1 = bold outline, 0 = normal Bit 1 : 1 = square-corner, 0 = round-corner
RadioItem	Bits 0-6 : family number (0 to 127)
ScrollBarItem	Bit 0 : 1 = up arrow on scroll bar Bit 1 : 1 = down arrow on scroll bar Bit 2 : 1 = left arrow on scroll bar Bit 3 : 1 = right arrow on scroll bar Bit 4 : 1 = horizontal scroll bar, 0 = vertical

NOTE: For all other items, set bit 7 of ItemFlag to 1 to make the item invisible.

The button is usually a rounded-corner rectangle, but it can also be a square-cornered rectangle with a drop shadow if bit 1 of ItemFlag is set to 1.

Most modal dialog boxes and alert boxes contain at least one button so that they can be dismissed in accordance with the user-interface guidelines.

A default button is one which may be selected by pressing the Return key. By convention, the default button is enclosed by a dark, black border so that it can be readily identified. If you define more than one button in a dialog box, the one most likely to be selected should be made the default so it can be selected easily from the keyboard.

For modal dialog boxes, a ButtonItem with an ID of 1 is always the default item. Such a button is typically labeled as the OK button. For alert boxes, a button with an ID of either 1 or 2 in the item list can be designated as the default when you create the alert's item list. These buttons are usually marked as the "OK" and "Cancel" buttons.

Check Boxes

A check box item (CheckItem) is always associated with a parameter that can be in one of two states: on and off, selected and not selected, high and low, and so on. A check box appears as a small square in the dialog box. When it is on, its value is 1, and it has an X drawn in it. When it is off, its value is 0, and the square is hollow.

The ItemDescr field in a check box template contains a pointer to the name of the check box. The name is drawn to the right of the check box, inside the item's display rectangle. ItemValue contains the initial value of the check box and can be either 1 (on) or 0 (off).

Radio Buttons

Radio button items (RadioItem) usually appear in groups of two or more, with each radio button representing a different value that may be associated with one particular parameter of interest to the application. They appear as small circles in a dialog box, and the one that is on (value 1) has a smaller black circle inscribed in it. The radio button's name, pointed to by ItemDescr, appears to the right of the button.

A radio button derives its name from the fact that when you select one by clicking it, all other buttons in the group are turned off, just like when you select a station on a standard car radio.

It is possible to assign associated radio buttons to the same family so that when one is turned on all the others will automatically turn off. To do this, put the family number (from 0 to 127) in bits 0 through 6 of ItemFlag.

Scroll Bars

Scroll bars have already been discussed in connection with the Window Manager. You can use them to reflect the value, in pictorial form, of any measurable quantity that might take on a range of values. For example, the position of the thumb in a vertical scroll bar might represent the amount of free space on a disk.

The ItemDescr field of a ScrollBarItem is a pointer to an action procedure that is called when any part of the scroll bar is selected. For information on how to design such a control, refer to the *Apple IIGS Toolbox Reference*. If the pointer is 0, the Dialog Manager does nothing special.

A scroll bar can take on values from 0 to 290; this range is reflected by the position of the thumb on the bar relative to the end points. ItemValue contains the initial value of the scroll bar.

The appearance of the control can be set using bits 0 to 4 of ItemFlag. These bits have the following meanings:

bit 0	1 = draw up arrow on scroll bar
bit 1	1 = draw down arrow on scroll bar
bit 2	1 = draw left arrow on scroll bar
bit 3	1 = draw right arrow on scroll bar
bit 4	1 = horizontal scroll bar, 0 = vertical

The ItemFlag value for a horizontal scroll bar with arrows would be $001C, for example

User Control Items

The first type of user control item (UserCtlItem) is a custom control designed in accordance with the specifications of the Control Manager. For these types of items, ItemDescr points to a control definition procedure, and ItemValue contains the initial value of the control. For instructions on how to design custom controls, refer to the Control Manager chapter of the *Apple IIGS Toolbox Reference*.

The other type of user control item (UserCtlItem2) is useful if you want to implement any control defined by the Control Manager. ItemDescr points to a parameter block that contains the address of the definition procedure for the control, a pointer to the title of the control, the view size, and the data size.

Static Text and Long Static Text

A regular static text item (StatText) is a string of up to 255 characters to be drawn in the dialog box. It is said to be static because you cannot edit it. A long static text item (LongStatText) is similar to a StatText item, but it can hold up to 32,767 characters. Both these items are used for such purposes as displaying commands, displaying explanatory messages, or posing questions.

For a StatText item, the ItemDescr field in the item template points to the string (preceded by a length byte) that contains the static text. The ItemValue field is undefined. For a LongStatText item, ItemDescr points to the start of the text (no preceding length byte). ItemValue contains the length of the string.

To define a static text item that is to be drawn on more than one line, you must insert carriage return codes (ASCII $0D) inside the text string. The Dialog Manager does not automatically wrap text at the item rectangle boundary.

Be careful to make the item rectangle deep enough to hold the text. If it is too shallow, the text will be clipped. If you are drawing a single line of text using the system font, the depth should be at least nine rows (the ascent+descent+leading of the font).

It is often convenient to be able to change the precise wording of a static text item after it is initially defined, by inserting filenames or phrases which cannot be predicted in advance. The easiest way to do this is to use the ^0, ^1, ^2, and ^3 text place holders when you first define the static text item.

You can assign a text string to each of these place holders to ensure that when the dialog box is drawn, the strings are substituted for the place holders. The instruction to use for this is _ParamText:

```
            PushPtr String0         ;String for ^0
            PushPtr String1         ;String for ^1
            PushLong #0             ;(Don't change ^2 string)
            PushPtr String3         ;String for ^3
            _ParamText

    String0  STR  'placeholder 0'
    String1  STR  'placeholder 1'
    String3  STR  '4th placeholder'
```

If you do not want to change a particular string, push a long word 0 on the stack instead of a pointer to the string. This was done for the ^2 place holder in the example.

Suppose you use a dialog box to ask for verification of a disk formatting operation. Instead of using a general static text item like "Are you sure you want to erase the disk?", you can define an item such as "Are you sure you want to erase ^0?" You can then use _ParamText to substitute for ^0 the actual name of the disk selected. This must be done before the dialog is displayed, of course.

Editable Line

An editable line item (EditLine) is a single line of text that may be edited using the standard editing techniques supported by the LineEdit tool set. The text can be up to 255 characters long and is enclosed by a rectangle.

You must be careful to ensure that the height of the rectangle for an EditLine item is at least two rows greater than the height of the font. If you are using the system font, for example, the rectangle height must be at least 11 rows. If the rectangle height is lower, no characters will appear.

The standard LineEdit editing techniques are as follows:

> An insertion point can be selected by clicking the mouse at any position in the text string. The insertion point is marked by a blinking vertical bar; subsequent characters will be placed at this location.

> A range of text can be selected by dragging the mouse across the text. (Selected text appears as white characters against a black background.) A selection range is deleted when a subsequent character is entered.
>
> A word can be selected by double-clicking the mouse anywhere in the word.
>
> All the text can be selected by triple-clicking.
>
> A selection range can be extended by clicking while holding down the Shift key.
>
> Pressing Control-X will erase the entire line.
>
> Pressing Control-Y will erase everything from the current insertion point to the end of the line. If a selection range is active, it is deleted.
>
> The left- and right-arrow keys can be used to move the insertion point back and forth in the text line.
>
> Pressing Control-F will delete the character at the insertion point or will delete the selection range.
>
> Pressing Delete will delete the character to the left of the insertion point and move all the characters to the right of the insertion point one position to the left, or will delete the selection range.
>
> Pressing any other key will insert that key character at the insertion point; if a range is selected, the characters in the range will be deleted and replaced by the key character.
>
> Open-Apple-X, Open-Apple-C, and Open-Apple-V can be used to perform cut, copy, and paste operations, all of which involve a data area called the clipboard. Cutting transfers the selection range to the clipboard and removes it from the line. Copying does the same thing, but without removing it from the line. Pasting involves transferring text from the clipboard to the line at the insertion point (if a selection range is active, the selection is deleted first).

Using an editable text item for user input makes it unnecessary for you to write your own line editor.

For an EditLine item, ItemDescr is a pointer to the string (preceded by a length byte) that will be shown when the dialog box first appears. The ItemValue field contains the maximum length of the string; the possible choices are 0 to 255.

When a dialog box is drawn, the first EditLine item is totally selected so that you can delete it quickly by pressing a key to enter a new string. If there is more than one editable text box item, you can use the Tab key to move from one to the next. If you are in the last text box when you press Tab, you will go to the first box.

As will be discussed later, the Dialog Manager contains functions you can use to change the EditLine text, to determine its contents, and to select any portion of it.

Icons

An icon (IconItem) is a static item that defines a small rectangular image on the screen. The ItemDescr field in a template contains a handle (not a pointer) to the icon definition. ItemValue is not used.

An icon definition begins with the pixel dimensions of the icon, in rectangle form. (The width of the icon must be a multiple of 8.) Following the rectangle are the bytes defining the bit image of the icon. When designing an icon, keep in mind that there are two bits per pixel in 640-by-200 mode and four bits per pixel in 320-by-200 mode.

Here is the definition for an icon in 640-by-200 mode that looks like an asterisk:

```
MyIcon ANOP
       DC      I'0,0,9,16'      ;Icon rectangle
       DC      H'FFF0FFFF'      ;Bit image defining each row
       DC      H'0FF0FF0F'      ;(2 pixels/nibble in 640 mode)
       DC      H'F0F0F0FF'
       DC      H'FF000FFF'
       DC      H'0000000F'
       DC      H'FF000FFF'
       DC      H'F0F0F0FF'
       DC      H'0FF0FF0F'
       DC      H'FFF0FFFF'
```

Remember that two 0 bits refer to a black pixel on the 640-by-200 screen, and two 1 bits refer to a white pixel.

Pictures

A picture item (PicItem) contains a handle to a QuickDraw II picture in its ItemDescr field. The ItemValue field is not defined.

User Item

A user item (UserItem) is one whose appearance and behavior is defined by the application. For such an item, ItemDescr points to an item-definition procedure and ItemValue is not used.

The Dialog Manager calls an item-definition procedure with a JSL instruction after first pushing two parameters on the stack: a pointer to the dialog (long) and the ID number of the item to draw (word). This information is provided so that you can use the same item-definition procedure to handle many user items and dialog windows.

After the procedure does what it is designed to do, perhaps drawing a fancy border or playing a tune, it must remove the input parameters from the stack and return. Here is the code fragment you will need to do this:

```
LDA     2,S      ;Remove parameters by moving
STA     8,S      ; the return address up
LDA     1,S      ; by 6 bytes
STA     7,S

TSC              ;Add 6 to the stack pointer
CLC
ADC     #6
TCS
RTL
```

The procedure ends with RTL because it is called with a JSL instruction.

DISABLING ITEMS

Any item in a dialog can be marked as disabled by adding the ItemDisable constant ($8000) to its item type code. Disabled items still look the same, but user activity in them is not reported to the application. It is a good idea to disable static items like icons, pictures, and text so that the application does not have to bother dealing with mouse activity in them.

ADDING ITEMS TO DIALOG BOXES

If you have created a dialog box with NewModalDialog, you need to add items to it before anything useful will appear on the screen. Even if you have used GetNewModalDialog with predefined items, you may want to add new items to react to changes in standard operating conditions.

To add an item, use either GetNewDItem or NewDItem. Using GetNewDItem is the easiest:

```
PushLong theDialog      ;Push dialog pointer
PushPtr  ItemTemplate   ;Push pointer to item template
_GetNewDItem
```

As you can see, GetNewDItem expects to find the item parameters in a template. The form of this template is the same as described in the previous section.

NewDItem expects you to push the item parameters on the stack instead:

```
PushLong theDialog      ;Dialog pointer
PushWord ItemID         ;Item ID number
PushPtr  ItemRect       ;Pointer to item rectangle
PushWord ItemType       ;Item type code
PushLong ItemDescr      ;Item descriptor
PushWord ItemValue      ;Item value
```

```
    PushWord ItemFlag        ;Item flag
    PushLong ItemColor       ;Ptr to item color table (0=default)
    _NewDItem
```

CHANGING ITEM ATTRIBUTES

Many Dialog Manager functions can be used to read and change attributes of the items in a dialog box item list.

GetIText	Return text of a StatText or EditLine item
SetIText	Set the text of a StatText or EditLine item
SelIText	Set the selection range of an EditLine item
GetDItemType	Get the item type code for an item
SetDItemType	Set the item type code for an item
GetDItemBox	Return the display rectangle for an item
SetDItemBox	Set the display rectangle for an item
GetDefButton	Return the ID of the default item
SetDefButton	Set the ID of the default item
GetDItemValue	Return the itemValue for an item
SetDItemValue	Set the itemValue for an item
HideDItem	Erase an item
ShowDItem	Display a hidden item
FindDItem	Return the ID of the item at a given point
DisableDItem	Disable an item
EnableDItem	Enable an item

Of these functions, only GetIText, SetIText, SelIText, GetDItemValue, and SetDItemValue are commonly used by applications. You may find a need for the others if you need to write a complex filter procedure (see the discussion of filter procedures below).

This section discusses ways of using this group of five major attribute commands. For information on the others, refer to the *Apple IIGS Toolbox Reference*.

Dealing with Text Items

It is often necessary to determine the text associated with a particular EditLine item so that the application can take the user's input and deal with it. For this, use GetIText:

```
            PushLong theDialog      ;Dialog pointer
            PushWord itemID         ;ID number of item
            PushPtr  theText        ;Pointer to string area
            _GetIText
            RTS

    theText DS      256             ;Space for returned string
```

The space you reserve for theText must be one byte greater than the maximum length of the text item permitted. The extra byte is for the leading length byte. You can also use GetIText with a StatText item if you wish.

To change the text of a StatText or EditLine item, use SetIText:

```
            PushLong theDialog        ;Dialog pointer
            PushWord itemID           ;Item ID number
            PushPtr  MyText           ;The new text string
            _SetIText

MyText  STR      'This is a new string'
```

The string at MyText must be preceded by a length byte. You can also change the text of an item using the ParamText command described earlier in this chapter. For this to work, the item text must contain place holder characters of the form ^0, ^1, ^2, and ^3.

Another thing you can do is preselect all or a portion of an EditLine item. Selected text is highlighted in white letters on a black background and is deleted and replaced when you type a character from the keyboard. If the item contains a default string, you will probably want to select the entire string, so the Dialog Manager does this for you. That way, the default will disappear when the user types a character to change the entry.

Suppose the user is to enter an EditLine item that ends with a filename suffix of ".ASM". If you display a default, you may wish to highlight everything but the suffix so that the user does not have to retype the suffix if he enters a new name. Here is how you arrange for the proper highlighting using SelIText:

```
PushLong theDialog        ;Dialog pointer
PushWord itemID           ;Item ID number
PushWord #0               ;Starting char. position
SEC
LDA      NameLen          ;Get length of name
SBC      #4               ;Subtract suffix size
PHA                       ;Ending character position
_SelIText
```

The starting and ending position parameters passed to SelIText run from 0 up to the maximum size of the text string. A 0 value refers to a position to the left of the first character, 1 refers to a position to the left of the second character, and so on. If the starting and ending positions are equal, a blinking vertical bar appears at that position. Typed characters are inserted at this point.

Reading and Changing the Item Value

To read the current numeric value of an item, use GetDItemValue as follows:

```
    PHA                         ;Space for result
    PushLong theDialog          ;The dialog pointer
    PushWord itemID             ;The ID of the item
    _GetDItemValue
    PLA                         ;Get the result
```

Of course, this function is useful only for those items that use the ItemValue field in the item template. For check boxes and radio buttons, the value can be 0 (off) or 1 (on). Other values are possible for scroll bar and user control items. For a LongStatText item, the value is the length of the text.

Use SetDItemValue to assign a particular value to an item. You will use it most often to select and deselect check boxes and radio buttons. For example, if you detect a mouse click in a check box that is off, you would turn it on (put an X in it) using the following instructions:

```
    PushWord #1                 ;New value (1=on)
    PushLong theDialog          ;Dialog pointer
    PushWord itemID             ;ID of dialog item
    _SetDItemValue
```

When you turn on a radio button item like this, the Dialog Manager automatically turns off all radio button items with the same family number. This is in keeping with the user-interface guidelines, which insist that only one radio button in a group may be on.

USING DIALOG BOXES

The proper way to handle a dialog box once it is on the screen depends on whether it is a modal or modeless dialog box.

Modal Dialog Boxes

As soon as you define a modal dialog box on the screen with NewModalDialog or GetNewModalDialog you must call ModalDialog to monitor events within the box and get a result indicating what item was selected. You can then deal with the result as you see fit before calling ModalDialog once again to get more input or dismissing the dialog box if a button was selected.

To dismiss a box, call CloseDialog to erase it from the screen. This function requires only one parameter, a dialog pointer.

To use ModalDialog, push space for a word result and then push the address of a filter procedure:

```
PHA                         ;Space for result
PushPtr FilterProc          ;Pointer to filter procedure
_ModalDialog
PLA                         ;ID of item selected
```

A filter procedure is a subroutine ModalDialog calls after it detects an event but before it responds to it. By using such a procedure, you can modify the effect of an event any way you like. If you are not using a custom filter procedure (the usual case), just push a long word zero on the stack. This invokes the standard filter procedure which converts a press of the Return key to a mouse click in the box's default button; it also handles the keyboard commands for cut (OpenApple-X), copy (OpenApple-C), and paste (OpenApple-V) operations properly.

A sample filter procedure is shown in listing 8–2. When ModalDialog calls it, there is a result space, three long values, and a three-byte return address on the stack:

```
result (word)
dialog pointer (long)
pointer to event record (long)
pointer to itemHit variable (long)
return address (3 bytes)
```

When the filter procedure ends, it must remove the three long input parameters from the stack by moving the return address up by 12 bytes and adjusting the stack pointer accordingly.

The result of a filter procedure is a Boolean value. If it is true, ModalDialog ends and returns the ID number which the filter procedure returns in itemHit. If it is false, ModalDialog handles the event in its usual way.

The filter procedure can tell what type of event it is dealing with by checking the "what" field of the event record. To force the event to be ignored, it can put a 0 in the What field to convert the event to a null event.

The procedure in listing 8–2 filters out keyboard control characters (other than those needed for editing) in this way. Without this filter, control characters appear as inverse question marks in an EditLine item.

When designing a custom filter procedure, you should strive to retain the behavior of the default filter procedure. To do this, set the high-order bit of FilterProc's address when you push it on the stack for ModalDialog. This tells ModalDialog to pass events to the standard filter after the custom filter procedure deals with them.

When ModalDialog takes over, it handles any update events related to the dialog (caused when a control item like a button or a check box changes value) and monitors all events until an active item is selected. It beeps if the mouse is clicked outside the dialog window and ignores all clicks inside the window if they are not also inside the display rectangle of an enabled item. Mouse-down events in control items, such as buttons, radio buttons, and check boxes, are monitored until the mouse is released; if the mouse is not still in the item upon release, the click is ignored.

When ModalDialog finishes it returns the number of the item selected. The application can then deal with this result as it wishes; it can then either remove the dialog box from the screen or call ModalDialog once again.

The action a program takes when ModalDialog returns a result depends on the type of item selected. If it was a check box, the box should be checked if it was previously unchecked or vice versa. If it was a radio button, the radio button should be selected and all other radio buttons in the group should be deselected. Standard subroutines for doing this are shown in listings 8–3 and 8–4.

When a button is selected, you should read and save the settings of the variable items in the dialog and then dismiss the dialog by removing its window from the screen with CloseDialog.

Key-down events are processed by ModalDialog only if there is an EditLine item in the dialog. If the EditLine item is enabled, its item number is returned after every key press. If it is not, no item number is returned, but ModalDialog still lets you edit the string in the text box. It is probably best to disable EditLine items (by adding $8000 to the item code in the item template) so that you do not have to keep looping back to ModalDialog after every key press. Instead, after a button is pressed to dismiss the dialog, you can use GetIText to determine the final value of the text string.

The Tab key is used to move the text insertion cursor from one EditLine item to the next. If you are in the last EditLine item when you press Tab, you will proceed to the first such item.

Mouse-down activity in EditLine items is automatically handled by ModalDialog in accordance with the LineEdit specifications described earlier. In fact, ModalDialog calls LineEdit to handle these events.

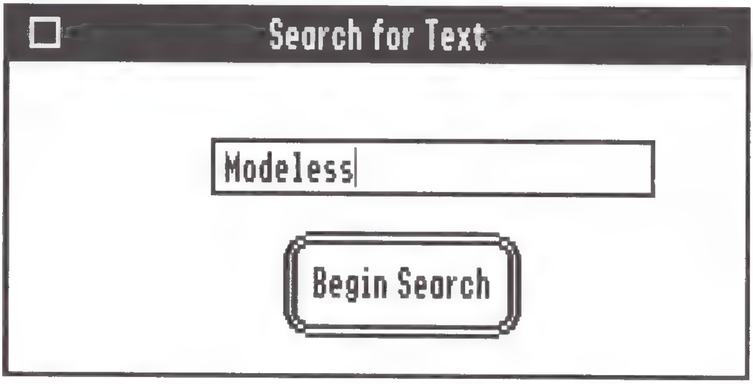
Modeless Dialog Boxes

A modeless dialog box is a bit more difficult to handle than a modal dialog box. When a modeless dialog box is on the screen, the user is not restricted from performing other operations (such as activating another window or selecting a desk accessory) before dismissing it to remove it from the screen. In this respect, a modeless dialog box is like any other standard window, although it contains various controls. Unlike a modal dialog box or an alert box, it does not retain control until you select an active item. You simply feed it events one at a time and it returns a Boolean result that tells you whether the event related to the dialog box or not.

To create a modeless dialog window, you must use NewModelessDialog as follows:

```
PLA                     ;space for result (long)
PLA
PushPtr dBoundsRect     ;Pointer to window rectangle
```

Figure 8–4. A Modeless Dialog Box



```
PushPtr dTitle             ;Pointer to dialog title
PushLong dBehind           ;Pointer to window in front
                           ; of dialog window
PushWord dFlag             ;Window frame bit vector
PushLong dRefCon           ;reference constant
PushPtr  dFullSize         ;Pointer to zoom rectangle
_NewModelessDialog
PopLong DialogPtr          ;pop pointer to dialog
```

Most of these parameters are self-explanatory. Normally, you set dBehind to -1 to bring the modeless dialog box up in front of all other windows. The dFlag vector has the same meaning as the window frame vector in a NewWindow call; most dialog windows have only a close box and a title area.

The parameters passed to NewModelessDialog do not include the items to be drawn in the dialog window. You must add these by making calls to GetNewDItem or NewDItem.

The program in listing 8–5 shows how to define a modeless dialog box. The box it defines is shown in figure 8–4.

The Dialog Manager makes it somewhat easier to deal with a modeless dialog box than a standard window. Whenever your program calls TaskMaster or GetNextEvent, it should call IsDialogEvent to determine whether the current event relates to the modeless dialog box (see the program in listing 8–6):

```
PHA                         ;Space for Boolean result
PushPtr EventRecord     ;TaskMaster/GetNextEvent record
_IsDialogEvent
PLA                         ;Pop true/false result
```

The EventRecord is the one used with TaskMaster or GetNextEvent. Listing 8–6 shows how to handle events in modeless dialog boxes.

If the result is false, the event was not dialog-related and can be processed as usual. This includes events for which TaskMaster returns wInMenuBar or wInSpecial, unless it was caused by a key-down event (that is, unless the user entered a keyboard equivalent).

Special edit items involving modeless dialog windows can be handled by calling a subroutine like the one in listing 8–7. In brief, it checks to see if the active window is the modeless dialog window; if it is, it handles the special edit item by calling one of the following functions:

DlgCut	Cut the selected text and put it on clipboard
DlgCopy	Copy the selected text to the clipboard
DlgPaste	Transfer the text on clipboard to the document
DlgDelete	Erase the selected text; clipboard not affected

(There is no similar function for handling the special Undo item. An application should handle Undo by canceling the previous edit operation.) All these dialog editing functions require only one parameter, the handle to the dialog record.

IsDialogEvent returns true for the following events:

- Activate or update events for a dialog window
- Mouse-down events in the content region of an active dialog window
- Any other event when a dialog window is active

When IsDialogEvent returns a true result, you normally call DialogSelect to process it. If you are using TaskMaster, however, you should check first to see if the TaskMaster result was wInMenuBar or wInSpecial. For either result, TaskMaster will have highlighted the menu title, so turn off highlighting with HiliteMenu (see chapter 7). DialogSelect does not do this for you.

The wInSpecial results should then be passed directly to DialogSelect for processing. If the modeless dialog box has an EditLine item, the item will be edited accordingly. The wInMenuBar results should be dealt with separately because DialogSelect will not know what to do with them.

Here is how to call DialogSelect:

```
         PHA                      ;Space for Boolean result
         PushPtr  EventRecord     ;Taskmaster/GetNextEvent recor
         PushPtr  theDialog       ;Dialog pointer returned here
         PushPtr  itemHit         ;Item number returned here
         _DialogSelect
         PLA                      ;Get true/false result
         RTS
```

```
    itemHit   DS  2                          ;Item number returned
    theDialog DS  4                          ;Pointer to dialog record
```

DialogSelect takes the event, processes it, and returns a Boolean result indicating whether it related to an enabled dialog item. If the result is false, it did not relate to an enabled item, and you do not have to do anything further. DialogSelect always returns a false value if you pass it a window update or activate event; these events are processed internally.

If the result is true, the itemHit and theDialog variables will contain the number of the item selected and a pointer to the active dialog record. You can then deal with the result in the same way you deal with a result returned by a call to ModalDialog for a modal dialog box. DialogSelect always returns a true result in situations in which ModalDialog would have reported an item-related event to you.

USING ALERT BOXES

There are four Dialog Manager functions for displaying alert boxes on the screen:

- Alert
- StopAlert
- NoteAlert
- CautionAlert

You call each of these functions in exactly the same way. The only differences among them are the icons they display in the top left corner of the box. Alert displays no icon at all; StopAlert displays an octagonal stop sign with a hand in it; NoteAlert displays a talking person; CautionAlert displays a yield sign with an exclamation mark in it.

The program in listing 8–8 shows how to create and deal with an alert box.

Alert boxes are easy to use, because all screen and event activities are handled by a single function. This function creates the alert record, draws the alert box and its items, interprets events until an active item is selected, erases the alert from the screen, and disposes of any memory used by the alert record. The function returns the item number selected. All you have to do is monitor this result and take whatever action is appropriate. Compare this with dialog boxes, which require you to use different functions to create and dispose of the dialog.

To display an alert box, pass pointers to an alert template and a filter procedure to the function:

```
    PHA                         ;space for result
    PushPtr AlertTemplate       ;Pointer to alert template
    PushPtr FilterProc          ;Pointer to filter procedure
    _Alert
    PLA                         ;ID of item selected
```

Alert handles mouse clicks much as ModalDialog handles them. That is, mouse clicks outside the alert box produce error beeps, and clicks in disabled items are ignored. Alert returns the value of any enabled item that is selected. For an alert box, only button items should be enabled.

An alert template, like a dialog template, includes a list of items that are to appear in the alert box. Its exact structure is as follows:

```
    DC      I't,l,b,r'          ;Alert box rectangle
    DC      I'AlertID'          ;ID number for alert box
    DC      I1'Stage1'          ;First stage alert
    DC      I1'Stage2'          ;Second stage alert
    DC      I1'Stage3'          ;Third stage alert
    DC      I1'Stage4'          ;Fourth stage alert
    DC      I4'Item1'           ;Pointer to 1st item template
    DC      I4'Item2'           ;Pointer to 2nd item template
    ...
    DC      I4'ItemN'           ;Pointer to Nth item template
    DC      I4'0'               ;zero terminator
```

The item templates referred to in an alert template are identical to the ones discussed earlier in connection with modal dialog boxes.

Stage bytes make up an important parameter in an alert template. They define the behavior of the alert in each of four different stages. When you use the alert box for the first time, it enters the first stage. As you keep calling it, it progresses through the second, third, and fourth stages, in that order. Thereafter, the alert box always behaves as if it was in the fourth stage. You can reset the alert stage to 0 with ResetAlertStage (no parameters).

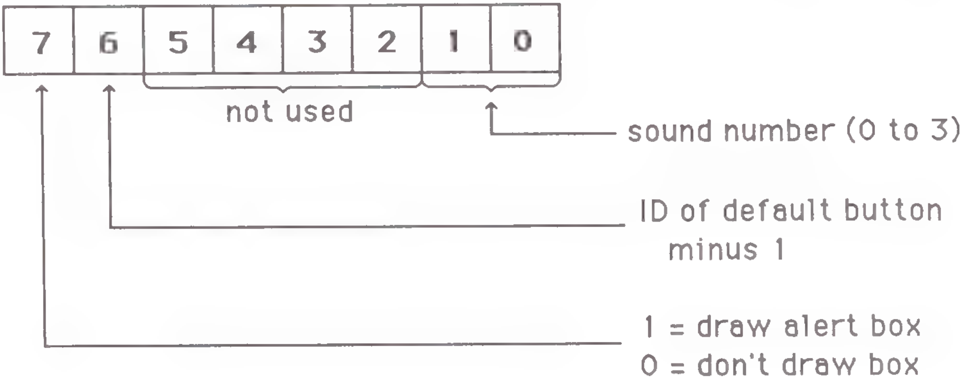
As shown in figure 8–5, the characteristics associated with each stage are what the default button is to be, what sound is to be emitted, and whether the alert box is to be drawn.

For a given stage, the first two bits (0 and 1) contain a sound number from 0 to 3. In most cases, this number represents the number of times the speaker is to beep when an alert is called up at that stage level. It is possible, however, to invoke a custom sound procedure that interprets these numbers differently. Refer to the *Apple IIGS Toolbox Reference* for instructions on how to do this.

Bit 7 indicates whether the alert box is to be drawn on the screen. You will usually set this bit to 1 (display the box), but you can set it to 0 if you do not want the alert to be displayed at that stage level.

Bit 6 controls which of two buttons is to be the default button. The default button is the one selected when Return is pressed from the keyboard. If the bit is set to

Figure 8–5. The Format of a Stage Byte in an Alert Template



0, the first button (usually an OK button) is the default; if it is 1, the second button (usually a Cancel button) is the default.

In most applications you will probably want all stages to be equivalent, so all four bytes will be the same. For example, if you want to beep the speaker once, display the alert box, and make the Cancel button the default button, use a stage byte of $C1. This sets the sound number to 01 (one beep), sets bit 7 to 1 (draw the box), and sets bit 6 to 1 (the default is #2, Cancel).

The filter procedure used by an alert box is similar to the one described earlier for dialog boxes. Pushing a 0 value tells the function to use the standard filter procedure. It converts a Return keypress into the click of the default button in the alert box.

REFERENCE SECTION

Table R8–1: The Major Functions in the Dialog Manager Tool Set ($15)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
Alert	$17	result (W)	ID of item selected
		AlertTemp (L)	Ptr to alert template
		FilterProc (L)	Ptr to filter procedure

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
CautionAlert	$1A	result (W)	ID of item selected
		AlertTemp (L)	Ptr to alert template
		FilterProc (L)	Ptr to filter procedure
CloseDialog	$0C	TheDialog (L)	Ptr to dialog record
DialogSelect	$11	result (W)	Boolean: enabled item chosen?
		EventRecord (L)	Ptr to event record
		TheDialog (L)	Ptr to space for dialog ptr
		ItemHitPtr (L)	Ptr to space for item number
DialogShutDown	$03	[no parameters]	
DialogStartup	$02	UserID (W)	ID tag for memory allocation
DisableDItem	$39	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of item to disable
DlgCut	$12	TheDialog (L)	Ptr to dialog record
DlgCopy	$13	TheDialog (L)	Ptr to dialog record
DlgDelete	$15	TheDialog (L)	Ptr to dialog record
DlgPaste	$14	TheDialog (L)	Ptr to dialog record
EnableDItem	$3A	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of item to enable
FindDItem	$24	result (W)	ID of item at point
		TheDialog (L)	Ptr to dialog record
		ThePoint (L)	Point to check (global)
GetDefButton	$37	result (W)	ID of default item
		TheDialog (L)	Ptr to dialog record

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
GetDItemBox	$28	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
		ItemBoxPtr (L)	Ptr to returned rectangle
GetDItemType	$26	result (W)	Item type code
		TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
GetDItemValue	$2E	result (W)	Value of dialog item
		TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
GetIText	$1F	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
		TheString (L)	Ptr to result string space
GetNewDItem	$33	TheDialog (L)	Ptr to dialog record
		ItemTemp (L)	Ptr to item template space
GetNewModalDialog	$32	result (L)	Ptr to dialog record
		DialogTemp (L)	Ptr to dialog template
HideDItem	$22	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
IsDialogEvent	$10	result (W)	Boolean: dialog-related?
		EventRecord (L)	Ptr to event record
ModalDialog	$0F	result (W)	ID of selected item
		FilterProc (L)	Ptr to filter procedure
NewDItem	$0D	TheDialog (L)	Ptr to dialog record
		ItemID (W)	Item ID number
		ItemRect (L)	Ptr to item rectangle
		ItemType (W)	Item type code
		ItemDescr (L)	Item descriptor

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
		ItemValue (W)	Item value
		ItemFlag (W)	Item flags
		ItemColorPtr (L)	Ptr to item color table
NewModalDialog	$0A	result (L)	Ptr to dialog record
		DBoundsRect (L)	Ptr to dialog rectangle
		DVisible (W)	Boolean: is it visible?
		DRefCon (L)	Reference constant
NewModelessDialog	$0B	result (L)	Ptr to dialog record
		DBoundsRect (L)	Ptr to window rectangle
		DTitlePtr (L)	Ptr to dialog title
		DBehindPtr (L)	Ptr to window in front
		DFlag (W)	Window frame bit vector
		DRefCon (L)	Reference constant
		DFullSizePtr (L)	Ptr to zoom rectangle
NoteAlert	$19	result (W)	ID of item selected
		AlertTemp (L)	Ptr to alert template
		FilterProc (L)	Ptr to filter procedure
ParamText	$1B	Param0Ptr (L)	Ptr to ^0 string
		Param1Ptr (L)	Ptr to ^1 string
		Param2Ptr (L)	Ptr to ^2 string
		Param3Ptr (L)	Ptr to ^3 string
SelIText	$21	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
		StartSel (W)	Start of selection range
		EndSel (W)	End of selection range
SetDefButton	$38	TheDialog (L)	Ptr to dialog record
		DefButID (W)	ID of new default button

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
SetDItemBox	$29	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
		ItemBoxPtr (L)	Ptr to new rectangle
SetDItemType	$27	ItemType (W)	New item type code
		TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
SetDItemValue	$2F	ItemValue (W)	New value for item
		TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
SetIText	$20	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
		TheString (L)	Ptr to new text string
ShowDItem	$23	TheDialog (L)	Ptr to dialog record
		ItemID (W)	ID of dialog item
StopAlert	$18	result (W)	ID of item selected
		AlertTemp (L)	Ptr to alert template
		FilterProc (L)	Ptr to filter procedure

Table R8–2: Dialog Manager Error Codes

Error Code	*Description of Error Condition*
$150A	The item type code is invalid.
$150B	The NewItem call was unsuccessful.
$150C	The specified item was not found.
$150D	The active window is not a modal dialog box.

Table R8–3: Useful Functions in the LineEdit Tool Set ($14)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
LEShutDown	$03	[no parameters]	
LEStartup	$02	UserID (W)	ID tag for memory allocation
		DPAddr (W)	Address of 1 page in bank 0

Listing 8–1: Defining and Handling a Modal Dialog Box

```
Modal      START
           Using   DlogData

; Create the modal dialog box:

           PHA                              ;Space for result
           PHA
           PushPtr DialogBox
           _GetNewModalDialog
           PopLong TheDialog

GoModal    PHA                              ;space for result
           PushLong #0                      ;standard filter procedure
           _ModalDialog                     ;Handle activity in box
           PLA                              ;Get the item selected

           CMP     #1                       ;OK button?
           BEQ     GoModal1                 ;Yes, so branch

           CMP     #20                      ;Checkbox?
           BCC     Toggle                   ;Yes, so branch

; If we reach here, it must be a radio button.

; When you turn one radio button on, all other family
; members are turned off automatically:

           TAX
           PushWord #1                      ;Turn selected radio button on
           PushLong TheDialog
           PHX                              ;Push selected item
           _SetItemValue                    ;Redraw with new value
           JMP     GoModal                  ;Go back to dialog
GoModal1   ANOP

; Here is where you would inspect and save the settings of
; all the variable parameters so that you can use them in
; another part of the program. Use GetItemValue and GetIText
; to determine the value for each item.

           PushLong TheDialog
           _CloseDialog                     ;Get rid of dialog
           RTS
```

```
; Toggle the setting of the check box:

Toggle      ANOP

            TAX
            PHX                         ;Save ID

            PHA                         ;Space for result
            PushLong TheDialog
            PHX                         ;Push item ID
            _GetItemValue
            PLA                         ;Get current value

            EOR     #$01                ;Toggle value bit

            PLX                         ;Get ID back
            PHA                         ;New value
            PushLong TheDialog
            PHX
            _SetItemValue               ;redraw with new value
            JMP     GoModal             ;Back to dialog

            END

DlogData    DATA

TheDialog   DS      4

; Dialog Template:

DialogBox   DC      I'30,100,135,465'
            DC      I'$FFFF'              ;True = visible
            DC      I4'0'                 ;refcon
            DC      I4'DItem1'
            DC      I4'DItem2'
            DC      I4'DItem3'
            DC      I4'DItem4'
            DC      I4'DItem5'
            DC      I4'DItem6'
            DC      I4'DItem7'
            DC      I4'DItem8'
            DC      I4'DItem9'
            DC      I4'DItem10'
            DC      I4'DItem11'
            DC      I4'DItem12'
            DC      I4'DItem13'
            DC      I4'DItem14'
            DC      I4'DItem15'
            DC      I4'DItem16'
            DC      I4'DItem17'
            DC      I4'DItem18'
            DC      I4'DItem19'
            DC      I4'DItem20'
            DC      I4'DItem21'
            DC      I4'0'

DItem1      DC      I2'1'                   ;Item ID (1 = default button)
            DC      I2'80,300,100,360'      ;Display rectangle (local)
            DC      I2'ButtonItem'          ;Item type code
            DC      I4'ButtonStr'           ;Name of button
            DC      I2'0'                   ;Not used
            DC      I2'0'                   ;Item flag (default)
            DC      I4'0'                   ;Color table ptr (default)

ButtonStr   STR     'OK'

DItem2      DC      I2'2'                   ;Item ID
            DC      I2'5,90,14,300'         ;Display rectangle (local)
            DC      I2'StatText+$8000'      ;Item type code (disabled)
            DC      I4'Stat1'               ;Static text string
            DC      I2'0'                   ;Item value (unused)
            DC      I2'0'                   ;Item flag (default)
            DC      I4'0'                   ;Color table ptr (default)

Stat1       STR     'Communications Parameters'    ;Static text string

DItem3      DC      I2'3'                   ;Item ID
            DC      I2'17,10,27,100'        ;Display rectangle (local)
            DC      I2'StatText+$8000'      ;Item type code (disabled)
            DC      I4'Stat2'               ;Static text string
            DC      I2'0'                   ;Item value (unused)
            DC      I2'0'                   ;Item flag (default)
            DC      I4'0'                   ;Color table ptr (default)

Stat2       STR     'Baud Rate:'            ;Static text string

DItem4      DC      I2'4'                   ;Item ID
            DC      I2'17,100,27,180'       ;Display rectangle (local)
            DC      I2'RadioItem'           ;Item type code
            DC      I4'Radio1'              ;Name of button
            DC      I2'0'                   ;Value: 0 = off / 1 = on
            DC      I2'1'                   ;Baud rate family
            DC      I4'0'                   ;Color table ptr (default)

Radio1      STR     '300'

DItem5      DC      I2'5'                   ;Item ID
            DC      I2'17,180,27,260'       ;Display rectangle (local)
            DC      I2'RadioItem'           ;Item type code
            DC      I4'Radio2'              ;Name of button
```

```
           DC     I2'1'              ;Value: 0 = off / 1 = on
           DC     I2'1'              ;Baud rate family
           DC     I4'0'              ;Color table ptr (default)

Radio2     STR    '1200'

DItem6     DC     I2'6'              ;Item ID
           DC     I2'17,260,27,340'  ;Display rectangle (local)
           DC     I2'RadioItem'      ;Item type code
           DC     I4'Radio3'         ;Name of button
           DC     I2'0'              ;Value: 0 = off / 1 = on
           DC     I2'1'              ;Baud rate family
           DC     I4'0'              ;Color table ptr (default)

Radio3     STR    '2400'

DItem7     DC     I2'7'              ;Item ID
           DC     I2'29,10,39,100'   ;Display rectangle (local)
           DC     I2'StatText+$8000' ;Item type code (disabled)
           DC     I4'Stat3'          ;Static text string
           DC     I2'0'              ;Item value (unused)
           DC     I2'0'              ;Item flag (default)
           DC     I4'0'              ;Color table ptr (default)

Stat3      STR    'Data Bits:'       ;Static text string

DItem8     DC     I2'8'              ;Item ID
           DC     I2'29,100,39,150'  ;Display rectangle (local)
           DC     I2'RadioItem'      ;Item type code
           DC     I4'Radio4'         ;Name of button
           DC     I2'1'              ;Value: 0 = off / 1 = on
           DC     I2'2'              ;Data bits family
           DC     I4'0'              ;Color table ptr (default)

Radio4     STR    '8'

DItem9     DC     I2'9'              ;Item ID
           DC     I2'29,150,39,200'  ;Display rectangle (local)
           DC     I2'RadioItem'      ;Item type code
           DC     I4'Radio5'         ;Name of button
           DC     I2'0'              ;Value: 0 = off / 1 = on
           DC     I2'2'              ;Data bits family
           DC     I4'0'              ;Color table ptr (default)

Radio5     STR    '7'

DItem10    DC     I2'10'             ;Item ID
           DC     I2'41,10,51,100'   ;Display rectangle (local)
           DC     I2'StatText+$8000' ;Item type code (disabled)
           DC     I4'Stat4'          ;Static text string
           DC     I2'1'              ;Item value (unused)
           DC      I2'0'                ;Item flag (default)
           DC      I4'0'                ;Color table ptr (default)

Stat4      STR     'Stop Bits:'         ;Static text string

DItem11    DC      I2'11'               ;Item ID
           DC      I2'41,100,51,150'    ;Display rectangle (local)
           DC      I2'RadioItem'        ;Item type code
           DC      I4'Radio6'           ;Name of button
           DC      I2'1'                ;Value: 0 = off / 1 = on
           DC      I2'3'                ;Stop bits family
           DC      I4'0'                ;Color table ptr (default)

Radio6     STR     '1'

DItem12    DC      I2'12'               ;Item ID
           DC      I2'41,150,51,200'    ;Display rectangle (local)
           DC      I2'RadioItem'        ;Item type code
           DC      I4'Radio7'           ;Name of button
           DC      I2'0'                ;Value: 0 = off / 1 = on
           DC      I2'3'                ;Stop bits family
           DC      I4'0'                ;Color table ptr (default)

Radio7     STR     '2'

DItem13    DC      I2'13'               ;Item ID
           DC      I2'53,10,63,100'     ;Display rectangle (local)
           DC      I2'StatText+$8000'   ;Item type code (disabled)
           DC      I4'Stat5'            ;Static text string
           DC      I2'1'                ;Item value (unused)
           DC      I2'0'                ;Item flag (default)
           DC      I4'0'                ;Color table ptr (default)

Stat5      STR     'Parity:'            ;Static text string

DItem14    DC      I2'14'               ;Item ID
           DC      I2'53,100,63,180'    ;Display rectangle (local)
           DC      I2'RadioItem'        ;Item type code
           DC      I4'Radio8'           ;Name of button
           DC      I2'1'                ;Value: 0 = off / 1 = on
           DC      I2'4'                ;Parity family
           DC      I4'0'                ;Color table ptr (default)

Radio8     STR     'None'

DItem15    DC      I2'15'               ;Item ID
           DC      I2'53,180,63,260'    ;Display rectangle (local)
           DC      I2'RadioItem'        ;Item type code
           DC      I4'Radio9'           ;Name of button
           DC      I2'0'                ;Value: 0 = off / 1 = on
```

```
            DC      I2'4'              ;Parity family
            DC      I4'0'              ;Color table ptr (default)

Radio9      STR     'Even'

DItem16     DC      I2'16'             ;Item ID
            DC      I2'53,260,63,340'  ;Display rectangle (local)
            DC      I2'RadioItem'      ;Item type code
            DC      I4'Radio10'        ;Name of button
            DC      I2'0'              ;Value: 0 = off / 1 = on
            DC      I2'4'              ;Parity family
            DC      I4'0'              ;Color table ptr (default)

Radio10     STR     'Odd'

DItem17     DC      I2'17'             ;Item ID
            DC      I2'65,10,75,120'   ;Display rectangle (local)
            DC      I2'CheckItem'      ;Item type code (disabled)
            DC      I4'Check1'         ;Static text string
            DC      I2'0'              ;Value: 0 = off / 1 = on
            DC      I2'0'              ;Item flag (default)
            DC      I4'0'              ;Color table ptr (default)

Check1      STR     'Filter'

DItem18     DC      I2'18'             ;Item ID
            DC      I2'65,120,75,230'  ;Display rectangle (local)
            DC      I2'CheckItem'      ;Item type code (disabled)
            DC      I4'Check2'         ;Static text string
            DC      I2'1'              ;Value: 0 = off / 1 = on
            DC      I2'0'              ;Item flag (default)
            DC      I4'0'              ;Color table ptr (default)

Check2      STR     'XON/XOFF'

DItem19     DC      I2'19'             ;Item ID
            DC      I2'65,230,75,340'  ;Display rectangle (local)
            DC      I2'CheckItem'      ;Item type code (disabled)
            DC      I4'Check3'         ;Static text string
            DC      I2'0'              ;Value: 0 = off / 1 = on
            DC      I2'0'              ;Item flag (default)
            DC      I4'0'              ;Color table ptr (default)

Check3      STR     'Line Delay'

DItem20     DC      I2'20'             ;Item ID
            DC      I2'79,10,89,115'   ;Display rectangle (local)
            DC      I2'StatText+$8000' ;Item type code (disabled)
            DC      I4'Stat6'          ;Static text string
            DC       I2'0'                 ;Item value (unused)
            DC       I2'0'                 ;Item flag (default)
            DC       I4'0'                 ;Color table ptr (default)

Stat6       STR      'Download File:'      ;Static text string

DItem21     DC       I2'21'                ;Item ID
            DC       I2'77,120,90,275'     ;Display rectangle (local)
            DC       I2'EditLine+$8000'    ;Item type code (disabled)
            DC       I4'EditString'        ;Editable text string
            DC       I2'15'                ;Maximum string length
            DC       I2'0'                 ;Item flag (default)
            DC       I4'0'                 ;Color table ptr (default)

EditString STR       'Messages'

            END
```

Listing 8–2: A Filter Procedure for ModalDialog

```
; This is a custom filter procedure for ModalDialog.
; On entry, the stack is configured as follows:
;
;           result:      WORD
;           theDialog:   LONG
;           theEvent:    LONG
;           itemHit:     LONG
;           returnAddr:  3 BYTES
;
; Return a TRUE result if ModalDialog is to exit with itemHit
; or a FALSE result if it is to process the event itself.
;
; To use a filter procedure, pass a pointer to it to ModalDialog.
; Set the high bit of the pointer to daisy chain to the standard
; filter which handles keyboard editing commands and converts a
; Return press to a click of the default button. Here's how to
; do this:
;
;     PHA
;     LDA   #^FilterProc     ;Address high
;     ORA   #$8000           ;Set high bit
;     PHA
;     LDA   #FilterProc      ;Address low
;     PHA
;     _ModalDialog
;
; This particular filter effectively removes all non-editing
; control characters so they won't appear as inverse question
; marks in an EditLine item.
```

```
FilterProc START

; these are direct page addresses after PHD/TSC/TCD:

OldDB      EQU     $01
OldDP      EQU     OldDB+1
ReturnAddr EQU     OldDP+2
itemHit    EQU     ReturnAddr+3
theEvent   EQU     itemHit+4
theDialog  EQU     theEvent+4
result     EQU     theDialog+4

           PHB                      ;Save data bank register

           PHK                      ;Allow absolute addressing
           PLB

           PHD                      ;Save direct page
           TSC
           TCD                      ;Align d.p. with stack

           LDA     [theEvent]       ;Get "what" from event record
           CMP     #3               ;Key down event?
           BNE     Exit             ;No, so branch
           LDY     #2
           LDA     [theEvent],Y     ;Get ASCII code
           AND     #$007F           ;Convert to std ASCII
           CMP     #$0020           ;Control character?
           BCS     Exit             ;No, so branch

;Filter all control chars, except CR or ones used by LineEdit:

           LDX     #0
CtrlLook   CMP     CtrlTable,X      ;Is it in the list?
           BEQ     Exit             ;Yes, so branch
           INX                      ;Move to next entry
           INX
           CPX     TblSize          ;At end of table?
           BNE     CtrlLook         ;No, so branch

           LDA     #0               ;Convert to "null" event
           STA     [theEvent]

Exit       LDA     #0               ;False result

Exit1      STA     result

           PLD                      ;Restore d.p.
           PLB                      ;Restore data bank
```

```
; Discard input parameters and move stack up by 12 bytes:

           LDA      2,S
           STA      14,S
           LDA      1,S
           STA      13,S

           TSC
           CLC
           ADC      #12
           TCS

           RTL

; This table contains all the control characters
; that are specially treated by LineEdit, and CR.

CtrlTable  DC       I'$06'           ;Delete char below cursor
           DC       I'$08'           ;Left-arrow (backspace)
           DC       I'$09'           ;Tab
           DC       I'$0D'           ;CR
           DC       I'$15'           ;Right-arrow (forward)
           DC       I'$18'           ;Control-X (erase line)
           DC       I'$19'           ;Control-Y (clear to EOL)
TblSize    DC       I'TblSize-CtrlTable'

           END
```

Listing 8–3: The CheckHit Subroutine to be Called When ModalDialog Returns a Check Box Item

```
;Call CheckHit in response to a hit in a dialog
; check box. CheckHit changes the value of the
; item from 0 to 1 (if it is off) or from 1 to 0
; (if it is on).
;
;On entry, A contains the ID number of the
; check box item in question.
;
;TheDialog is a dialog pointer stored in
; the GlobalData data segment.

CheckHit   START
           Using    GlobalData

           TAX
           PHX                       ;Save ID
           PHA                      ;Space for result
           PushLong TheDialog
           PHX                      ;Push item ID
           _GetItemValue
           PLA


           EOR     #$01             ;Invert value bit


           PLX                      ;Get ID back
           PHA                      ;New value
           PushLong TheDialog
           PHX
           _SetItemValue            ;redraw with new value
           RTS


           END
```

Listing 8–4: The RadioHit Subroutine to be Called When ModalDialog Returns a Radio Button Item

```
;Call RadioHit in response to a hit in a dialog
; radio button. RadioHit sets the value of the
; item to 1. This causes all other family members
; to be turned off.
;
;On entry, A contains the ID number of the
; radio button item in question.
;
;TheDialog is a dialog pointer stored in
; the GlobalData data segment.


RadioHit   START
           Using   GlobalData


           TAX                      ;Save item ID in X
           PushWord #1              ;1 = on
           PushLong TheDialog
           PHX                      ;Push item ID
           _SetItemValue            ;redraw with new value
           RTS


           END
```

Listing 8–5: How To Create a Modeless Dialog Box

```
; Create a modeless dialog box:

ModeLess   START
           Using   DlogData

           PHA                            ;Space for result
           PHA
           PushPtr DialogRect             ;Content rectangle
           PushPtr DlogTitle              ;Title of window
           PushLong #-1                   ;Ptr to window in front (-1 = top)
           PushWord #%1100000010100000  ;frame: close, title, drag, visibl
           PushLong #0                    ;refcon
           PushLong #0                    ;zoom rectangle (0=default)
           _NewModelessDialog
           PopLong TheDialog

           PushLong TheDialog
           PushPtr DItem1
           _GetNewDItem

           PushLong TheDialog
           PushPtr DItem2
           _GetNewDItem
           RTS

           END

DlogData   DATA

DialogRect DC      I'30,60,100,425'
DlogTitle  STR     'Search for Text' ;title

TheDialog  DS      4                  ;Pointer to modeless dialog
; Dialog Template:

DItem1     DC      I2'1'              ;Item ID (1 = default button)
           DC      I2'40,140,60,250' ;Display rectangle (local)
           DC      I2'ButtonItem'     ;Item type code
           DC      I4'ButtonStr'      ;Name of button
           DC      I2'0'              ;Not used
           DC      I2'0'              ;Item flag (default)
           DC      I4'0'              ;Color table ptr (default)

ButtonStr  STR     'Begin Search'

DItem2     DC      I2'2'              ;Item ID
           DC      I2'17,100,30,320' ;Display rectangle (local)
           DC      I2'EditLine+$8000' ;Item type code (disabled)
           DC      I4'EditString'     ;Editable text string
           DC       I2'20'            ;Maximum string length
           DC       I2'0'             ;Item flag (default)
           DC       I4'0'             ;Color table ptr (default)

EditString STR      ''                ;Start with null string

           END
```

Listing 8–6: How To Handle Events in Modeless Dialog Boxes

```
; Use this type of event loop when a modeless
; dialog may be active.

EvtLoop    START
           Using    DlogData

           PHA
           PushWord #$FFFF            ;All events
           PushPtr  TaskRec
           _TaskMaster

           PHA                        ;space for result
           PushPtr  TaskRec
           _IsDialogEvent             ;Is it for modeless dialog?
           PLA                        ;Get Boolean
           BEQ      EvtLoop1          ;No, so branch

; If the keypress was a menu item keyboard equivalent, turn the
; menu highlighting off. DialogSelect doesn't do this for you.

           PLA                        ;Get the TaskMaster result
           CMP      #wInMenuBar       ;Standard menu item?
           BEQ      TitleOff          ;Yes, so fix menu title
           CMP      #wInSpecial       ;Special menu item?
           BNE      DoModeless        ;No, so do nothing

TitleOff   PushWord #0                ;Highlighting off
           PushWord TaskData+2        ;Get menu ID
           _HiliteMenu

           LDA      TaskData          ;Get menu item ID
           CMP      #256              ;Special editing item?
           BCC      DoModeless        ;Yes, so give it to DialogSelect

; Handle non-editing keyboard equivalents here,
; then return to the event loop.

           BRL      EvtLoop

DoModeless PHA                        ;space for Boolean result
           PushPtr  TaskRec           ;Task record pointer
           PushPtr DlogActive         ;Dialog pointer returned here
           PushPtr ItemHit            ;Item number returned here
           _DialogSelect
           PLA                        ;Was it handled?
           BEQ     EvtLoop            ;No, so branch

; The event is dialog-related if we reach here. DoDialog is a
; subroutine you would write to handle dialog activity. It can
; get the relevant dialog pointer from DlogActive and the item
; number from ItemHit.

           JSR     DoDialog           ;Handle dialog activity
           BRL     EvtLoop

EvtLoop1   PLA                        ;Get TaskMaster result

;          [handle TaskMaster result in usual way]

           BRL     EvtLoop            ;Back to event loop

DoDialog   ANOP
;          [this subroutine handles dialog selections]
           RTS

           END

DlogData   DATA

DlogActive DS      4
ItemHit    DS      2

EventRec   ANOP                       ;Event record
What       DS      2                  ;Event code
Message    DS      4                  ;Event result
When       DS      4                  ;Ticks since startup
Where      DS      4                  ;Mouse location (global)
Modifiers  DS      2                  ;Status of modifier keys
TaskData   DS      4                  ;TaskMaster data
TaskMask   DC      I4'$00001FFF'      ;TaskMaster handles all

           END
```

Listing 8–7: The Type of Subroutine a Program Should Call To Handle Editing When a Modeless Dialog Box is Active

```
; This subroutine handles special edit items for
; modeless dialog boxes. If the carry flag is set on
; exit, the edit command was handled.
;
; The pointer to the dialog is stored at TheDialog.
```

```
DoEditing  ANOP

; Check to see if the modeless dialog box is active:

           PHA
           PHA
           _FrontWindow                 ;Get pointer to active window
           PLA
           PLX                          ;Compare active window with
           CMP     TheDialog            ; dialog window.
           BNE     NoDlogEdit
           CPX     TheDialog+2
           BNE     NoDlogEdit

;Pass control to the appropriate edit item handler:

           SEC
           LDA     TaskData             ;Get menu ID
           SBC     #250                 ;Convert 250-254 to 0-4
           ASL     A                    ;x2 to step into table
           TAX
           JSR     (EditTable,X)

           SEC                          ;SEC means "was handled"
           RTS

NoDlogEdit CLC                          ;CLC means "not for dialog"
           RTS

EditTable  DC      I'DUndo'             ;item 250 (Undo)
           DC      I'DCut'              ;item 251 (Cut)
           DC      I'DCopy'             ;item 252 (Copy)
           DC      I'DPaste'            ;item 253 (Paste)
           DC      I'DClear'            ;item 254 (Clear)

DUndo      RTS                          ;(there is no standard undo)

DCut       PushLong TheDialog
           _DlgCut
           RTS

DCopy      PushLong TheDialog
           _DlgCopy
           RTS

DPaste     PushLong TheDialog
           _DlgPaste
           RTS

DClear     PushLong TheDialog
           _DlgDelete
           RTS
```

Listing 8–8: Defining and Handling an Alert Box

```
; Bring up the alert box and wait for a button push:

DoAlert     START
            Using   AlertData

            PHA                         ;space for result
            PushPtr  AlertBox
            PushLong #0                 ;0=default filter procedure
            _Alert
            PLA                         ;Get item ID
            RTS

AlertData   DATA

; Alert box template:

AlertBox    DC      I'60,70,120,400'    ;Alert rectangle
            DC      I'1'                ;Alert ID
            DC      I4'$C3C2C1C0'       ;Stages bytes
            DC      I4'AlertItem1'      ;Pointer to item #1
            DC      I4'AlertItem2'      ;Pointer to item #2
            DC      I4'AlertItem3'      ;Pointer to item #3
            DC      I4'0'               ;Terminator

AlertItem1  DC      I2'1'               ;Item ID
            DC      I2'30,80,50,140'    ;Display rectangle (local)
            DC      I2'ButtonItem'      ;Item type code
            DC      I4'OKText'          ;Name of button
            DC      I2'0'               ;Item value (off)
            DC      I2'0'               ;Item flag (default)
            DC      I4'0'               ;Color table ptr (default)

OKText      STR     'OK'

AlertItem2  DC      I2'2'               ;Item ID
            DC      I2'30,180,50,240'    ;Display rectangle (local)
            DC      I2'ButtonItem'      ;Item type code
            DC      I4'CancelText'      ;Name of button
            DC      I2'0'               ;Item value (off)
            DC      I2'0'               ;Item flag (default)
            DC      I4'0'               ;Color table ptr (default)

CancelText  STR     'Cancel'

AlertItem3  DC      I2'3'               ;Item ID
            DC      I2'10,80,22,320'    ;Display rectangle (local)
            DC      I2'StatText+$8000'   ;Item type code (disabled)
            DC      I4'StatString'      ;Static text string
            DC      I2'0'               ;Item value (unused)
            DC      I2'0'               ;Item flag (default)
            DC      I4'0'               ;Color table ptr (default)

; ^0 is a placeholder for a filename (Use ParamText to fill it in).

StatString STR      'Do you want to erase ^0?'   ;Static text string

            END
```

CHAPTER 9

All about Desk Accessories

Desk accessories are programs (usually small utilities) that reside in memory at the same time as a primary GS application. If the application is properly designed, a user can call up a desk accessory at any time, use it, return to the application, and continue on as if the application had never been interrupted. It is not necessary to restart the application from the beginning.

Popular desk accessories are programs that provide features that few applications support directly: a calculator, note pad, clock, and appointment calendar, for example.

The GS supports two types of desk accessories: classic desk accessories (CDAs) and new desk accessories (NDAs). As will be discussed, CDAs are available to any application but NDAs work only with applications which take advantage of the desktop environment..

To call up a CDA, press three keys on the keyboard simultaneously—Control-OpenApple-Esc. This causes an interrupt signal, which the GS services by clearing the text screen and displaying a menu containing a list of the names of up to fourteen CDAs to choose from. You can activate a specific CDA by using the arrow keys to highlight the CDA name and then pressing Return. There are two standard CDAs in ROM: the Control Panel (used to configure the system hardware and set preferences) and the Alternate Display Mode (used to enable or disable software shadowing of the secondary text screen from $800 to $BFF).

You can access the CDA menu any time 65816 interrupts are not disabled (with an SEI instruction), which is most of the time. Thus, CDAs can be used with any GS application, including traditional IIe-style applications and the new-style applications that work in the desktop environment.

NDAs, on the other hand, are available only to applications that use the desktop environment supported by the GS tools. This is because an NDA is called up by pulling down an "Apple" menu in the menu bar (created by the Menu Manager) and selecting the NDA's name. Most NDAs create windows that may coexist on the screen with windows created by the application; the user can switch between desk accessory windows (also called *system windows*) and application windows in the usual way—by clicking in the window to be activated.

The purpose of this chapter is to show how to write programs which will work properly with desk accessories and how to write desk accessory programs themselves. This involves a study of a tool set called the Desk Manager (tool set 5).

Before specifics are discussed, a preliminary word about the Desk Manager is appropriate. To start it up so that you can use desk accessories, call the DeskStartup function. Shut it down with the DeskShutDown function. Neither function requires parameters nor generates results.

CLASSIC DESK ACCESSORIES

An application does not have to do much to support classic desk accessories. It just cannot disable interrupts with an SEI instruction for long periods of time. When interrupts are disabled, the GS ignores the Control-OpenApple-Esc keyboard sequence, so the CDA menu will not appear.

Some programs may need to disable interrupts, but usually only for brief periods of time. If you are writing such a program, be sure to execute a CLI (clear interrupt flag) instruction once it is safe to re-enable interrupts.

Writing a CDA

A CDA is actually a ProDOS 16 load file that begins with a special header block containing the name of the CDA, the address of the subroutine to be called to start it up, and the address of the subroutine to be called when DeskShutDown is called. Here is the assembly-language format of the header:

```
CDA_Header STR  'CDA Name'      ;Name (preceded by length)
           DC   I4'CDA_Startup' ;Startup subroutine
           DC   I4'CDA_ShutDwn' ;ShutDown subroutine
```

The start-up subroutine contains the main code for the CDA. The Desk Manager calls it with a JSL instruction in full native mode when you select the CDA from the CDA menu. The subroutine must end with an RTL instruction.

The Desk Manager calls the shut-down subroutine whenever the DeskShutDown function is called. An application normally calls this function just before it quits, but it is also called when the operating system switches between ProDOS 16 and ProDOS 8. The CDA's shutdown subroutine can terminate any background task the accessory may be performing for the current application. In most cases, there are no such tasks, so the shutdown subroutine is just a single RTL instruction.

A CDA can use any of the text screen areas in banks $00, $01, $E0, $E1 (offsets $400-$7FF) without fear of interfering with the current application. This is because the Desk Manager saves these areas before displaying the CDA menu and restores them on exit. A CDA may also use zero page (page $00 of bank $00) without bothering to preserve its contents as long as it respects those areas used by the system monitor subroutines. Any other memory the CDA needs must be allocated with the Memory Manager.

If the CDA is not designed to work with all operating systems, it should check location $E100BC to see which one is active and then take the appropriate action. The number stored there is $00 for ProDOS 8 or $01 for ProDOS 16.

Installing a CDA

The CDA program file must have a file type code of $B9, and you must place it in the SYSTEM/DESK.ACCS/ subdirectory of the disk from which you boot. When you boot ProDOS 16, the system automatically installs the CDAs it finds in this subdirectory. The maximum number of CDAs that may be installed (including the Control Panel and Alternate Display Mode accessories in ROM) is fourteen.

If the CDA is already in memory, you can install it by passing its handle to InstallCDA. You probably will not use InstallCDA often because it is more convenient to have CDAs installed automatically at boot time.

NEW DESK ACCESSORIES

New Desk Accessories are more difficult for an application to support because they are more intimately connected to an application. Whereas a CDA takes complete control of the GS until you exit it, an NDA shares the screen with the application and will not work properly without a helping hand from the application.

To support NDAs, several tool sets must be loaded and started up before you call the DeskStartup function: QuickDraw, Event Manager, Window Manager, Menu Manager, Control Manager, Scrap Manager, LineEdit, and Dialog Manager. This is necessary because an NDA must be permitted to call functions in these tool sets without having to load RAM-based tool sets or having to execute start-up functions.

The easiest way to make NDAs available to an application is to use a TaskMaster event loop, because TaskMaster makes almost all necessary function calls for you. In fact, assuming that the application calls DeskStartup at the beginning and DeskShutDown at the end, it need only use FixAppleMenu to add the names of all the NDAs to a standard Apple menu:

```
    PushWord #1                ;ID number of Apple menu
    _FixAppleMenu
```

The NDA items are assigned consecutive menu item ID numbers, beginning with 1. (Recall from chapter 7 that numbers 1 through 249 are reserved for desk accessory items.)

If you are not using TaskMaster, NDA support is more complicated and involves calling specific Desk Manager functions at appropriate times. The following paragraphs describe how TaskMaster controls NDAs and what standard actions have to be performed if a GetNextEvent event loop were used.

TaskMaster knows when the user selects a desk accessory item from the Apple menu because MenuSelect returns a menu ID less than 250. When this occurs, TaskMaster calls OpenNDA to open the desk accessory, an action that usually causes a window for the NDA to appear on the screen. It then controls all window management for both desk accessory and application windows, just as it would if multiple application windows were on the screen.

TaskMaster handles mouse-down events in the system window content region by calling SystemClick. SystemClick passes the event to the NDA for processing. Standard editing events selected from the Edit menu (Undo, Cut, Copy, Paste, and Clear with menu IDs from 250 to 254) are handled by calling SystemEdit, which passes them to the active DA for processing.

A special close item in a File menu (menu ID 255) is handled by calling CloseNDAbyWinPtr. This passes control to the close subroutine defined in the NDA.

TaskMaster also allows NDAs to perform any periodic activity associated with them by calling SystemTask once every time it performs its internal event loop.

Writing an NDA

Like a CDA, an NDA is a ProDOS 16 load file that begins with a special header block. The format of the header block is quite different than that for a CDA, however. The header-block format looks like this:

```
Period      EQU   60               ;# of ticks between run actions
EventMask   EQU   $FFFF            ;mask describing events wanted

NDA_Header  DC    I4'NDA_Open'     ;Pointer to open subroutine
            DC    I4'NDA_Close'    ;Pointer to close subroutine
            DC    I4'NDA_Action'   ;Pointer to action subroutine
            DC    I4'NDA_Init'     ;Pointer to init subroutine
            DC    I2'Period'
            DC    I2'EventMask'

            DC    C'##'            ;Placeholder for length
            DC    C'Item Name'     ;Text for the NDA name
            DC    C'\H**',I1'0'    ;Space of ID + zero byte
```

As you can see, the header begins with pointers to four standard subroutines. They will be described below.

Following the pointers is a word indicating how often the Desk Manager will pass a Run code to the NDA_Action subroutine. The basic unit of Period is a timer tick (1/60th of a second), so, for example, a Period of 60 would represent one second.

There are two special Period values: 0 tells the Desk Manager to pass the Run code as often as possible (in practice, this means once every TaskMaster loop); $FFFF tells the Desk Manager not to pass the Run code at all.

The event mask indicates which events will cause the Desk Manager to pass an Event code to the NDA_Action subroutine. The meaning of the mask is the same as that for the event mask described in chapter 5 in connection with the GetNextEvent and TaskMaster functions. In most cases you will set it to $FFFF (all events).

The final item in the header is the name of the NDA, in the format expected by the Menu Manager. It begins with two placeholders for the length and is followed by the text of the name and then a \H** command (this reserves space for the menu ID). A trailing 0 byte marks the end of the item definition.

The next section takes a close look at each of the four subroutines an NDA must contain. You may want to refer to listing 9–1, which shows how to implement a clock desk accessory, for practical implementation instructions.

NDA_Open. The NDA_Open subroutine must prepare the desk accessory for use, but only if it is not already open. In most cases this involves creating a window for the NDA using the NewWindow function, saving the window pointer for later use, and marking the window as a system window using the SetSysWindow function.

Before the Desk Manager makes a call to NDA_Open with a JSL instruction, it reserves space on the stack for a long word result. NDA_Open must return the NDA's window pointer here before exiting with an RTL instruction.

On entry to the NDA_Open subroutine (or the other three standard subroutines), the 65816 will be in full native mode. At the start of the subroutine, you should execute PHB, PHK, and PLB instructions to make the data bank register equal to the code bank register. This lets you access memory locations within the desk accessory code space with absolute addressing rather than absolute long addressing. On exit, be sure to execute a PLB instruction to restore the data bank register to its original state.

NDA_Close. The NDA_Close subroutine is responsible for shutting down the accessory, usually by closing its window with the CloseWindow function. It does not return a result. This subroutine is automatically called by TaskMaster if you click the close box on the NDA window or if you select the special Close item from a menu (it has a menu ID of 255).

NDA_Init. This initialization subroutine is called whenever the application calls the DeskStartup or DeskShutDown function. On entry, the accumulator is 0 if a call to DeskShutDown was made; it is non-zero for calls to DeskStartup.

Most initialization subroutines do not have to do anything when DeskStartup is called, because the NDA is not even integrated with the application at this stage.

Table 9–1: Action Codes for New Desk Accessories

Action Code	*Task*
1	Event A mouse-down, mouse-up, key-down, autokey-down, update, or activate event took place, so process it. A pointer to the event record is in X (low) and Y (high).
2	Run The Period has elapsed, so perform the periodic action.
3	Cursor The NDA window is active, so change the cursor if necessary.
4	[Reserved]
5	Undo The special Undo item (ID 250) was selected, so process it.
6	Cut The special Cut item (ID 251) was selected, so process it.
7	Copy The special Copy item (ID 252) was selected, so process it.
8	Paste The special Paste item (ID 253) was selected, so process it.
9	Clear The special Clear item (ID 254) was selected, so process it.

When DeskShutDown is called, the NDA window should be closed to release the memory it occupies or to terminate some activity tied to the application.

NDA_Action. The most difficult subroutine to program is the action subroutine. It is responsible for handling any of nine tasks that may be passed to it for processing. The action code for the task is passed in the accumulator and can be one of the values shown in table 9–1.

For action codes 5 to 9 (the editing commands), a non-zero value is returned in the accumulator if the action was handled (the usual case) or 0 if it was not.

The Run task should be processed by performing the periodic action associated with the NDA. The clock accessory in listing 9–1, which has a Period of one second, processes the Run code by displaying the current time. In other types of accessories, you may want to blink a cursor, update a system status display, or perform other similar tasks.

The Cursor task is called once every TaskMaster loop, but only if the NDA window is open. It allows the accessory to change the active cursor depending on the current position of the mouse. For example, you might want to change the cursor to a bulls-eye if the mouse happens to be inside the content region of the desk accessory window. The example accessory shows how to determine if the mouse is inside an NDA window, how to change the cursor to a wristwatch if it is, and how to restore the original cursor if it is not.

The Event task handles the six standard GS events mentioned above. The trickiest handler is probably the one for update events. It must call BeginUpdate (to make the visible region the same as the update region temporarily), redraw the contents of the window, and then call EndUpdate (to restore the original visible region and empty the update region). The clock NDA redraws the window simply by displaying the current time.

The handler for activate events must first distinguish between an activate and a deactivate operation. It can do this by examining bit 0 of the modifiers word of the event record; if it is 0, the NDA window is being deactivated. In the example, a deactivate event is handled by restoring the original cursor.

Notice the technique used in the example for accessing the modifiers field in the event record. On entry to the NDA_Action subroutine, a pointer to the event record (stored in X and Y) is pushed on the stack. Later, a direct page that is aligned with the stack pointer is created so that the fields in the event record, a pointer to which is now in direct page, can be accessed with the [dp],Y addressing mode.

Installing an NDA

NDAs are ProDOS 16 load files that have a file type code of $B8. Only those NDA files that are located in the SYSTEM/DESK.ACCS/ subdirectory on the boot disk at boot time will be added to the Apple menu when you call the FixAppleMenu function.

REFERENCE SECTION

Table R9–1: The Major Functions in the Desk Manager Tool Set ($05)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
CloseNDA	$16	RefNum (W)	NDA reference number
CloseNDAbyWinPtr	$1C	TheWindow (L)	Pointer to CDA window
DeskStartup	$02	[no parameters]	
DeskShutDown	$03	[no parameters]	

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
FixAppleMenu	$1E	MenuID (W)	ID of menu to add DAs to
InstallCDA	$0F	IDHandle (L)	Handle to CDA header
OpenNDA	$15	result (W)	Reference number for NDA
		DAIDNumber (W)	Menu item ID number for NDA
SystemClick	$17	EventRecord (L)	Ptr to event record
		TheWindow (L)	Ptr to NDA window record
		FindWindRes (W)	Result of FindWindow call
SystemEdit	$18	result (W)	Boolean: did NDA handle edit?
		EditType (W)	Code for edit command: 1 = undo, 2 = cut, 3 = copy, 4 = paste, 5 = clear
SystemTask	$19	[no parameters]	

Table R9–2: Desk Manager Error Codes

Error Code	*Description of Error Condition*
$0510	The specified desk accessory was not found.
$0511	The window is not an NDA (system) window.

Listing 9–1: A New Desk Accessory

```
********************************************
* This program shows how to construct a    *
* New Desk Accessory (NDA). The NDA        *
* defined here is a clock window which     *
* contains the current time and date; the  *
* time and date are updated once per       *
* second.                                  *
*                                          *
********************************************
           LIST    OFF
           SYMBOL  OFF
           ABSADDR ON
           INSTIME ON
           GEN     ON

           KEEP    CLOCK                ;Code file

           MCOPY   CLOCK.MAC            ;Macro file

Period     GEQU     60                  ;Ask for "run" action every second
EventMask  GEQU     $FFFF               ;Handle all events

NDA_Clock  START

           DC      I4'NDA_Open'         ;Open the NDA
           DC      I4'NDA_Close'        ;Close the NDA
           DC      I4'NDA_Action'       ;Perform NDA action
           DC      I4'NDA_Init'         ;Startup/Shutdown the NDA
           DC      I2'Period'           ;Periodicity of "run" action
           DC      I2'EventMask'        ;Permitted events
           DC      C'#'                 ;Name in menu item form
           DC      C'Clock'             ;Text for NDA name
           DC      C'\H**',I1'0'        ;ID field + terminator

; Open the NDA if it has not been previously opened. This routine
; must return a pointer to the NDA window on the stack, just above
; the 3-byte return address. The Desk Manager reserves this result
; space just before calling NDA_Open with a JSL instruction.

NDA_Open   ANOP

Result     EQU     $05                 ;Result stack offset after JSL, PHB

           PHB
           PHK
           PLB                          ;data bank = code bank

           LDA     ClockOpen            ;Clock window already open?
           BNE     Ignore               ;if so, branch
           PHA                       ;Space for result
           PHA
           PushPtr WindowDef
           _NewWindow                ;Create and open NDA window

           PLX                       ;Pop pointer (low)
           PLA                       ;Pop pointer (high)

           STX     WindowPtr         ;Save pointer to window
           STA     WindowPtr+2

           STA     Result+2,S        ;Save result on stack (high)
           TXA
           STA     Result,S          ; (low)

           PushLong WindowPtr
           _SetSysWindow             ;Mark this as a DA window

           LDA     #$FFFF
           STA     ClockOpen         ;Set "open" flag

           PHA                       ;space for result
           PHA
           _GetCursorAdr
           PopLong OldCursor         ;Save pointer to regular cursor

Ignore     PLB
           RTL


; Close the NDA if it is not already closed:

NDA_Close  ANOP

           PHB
           PHK
           PLB                       ;data bank = code bank

           LDA     ClockOpen         ;Is the clock window open?
           BEQ     Ignore            ;No, so branch

           PushLong WindowPtr
           _CloseWindow              ;Get rid of the window

           STZ     ClockOpen         ;Mark clock as closed

           PushLong OldCursor
           _SetCursor                ;Restore application cursor

           PLB
           RTL
```

```
; Perform the NDA action:

NDA_Action ANOP

          PHB                        ;Save data bank
          PHK
          PLB                        ;Make data bank = program bank

          PHY                        ;Save incoming parameters
          PHX                        ;(event record or menu info)

          ASL     A                  ;x2 to step into table
          TAX
          JSR     (ActionTbl,X)

          PLX                        ;Fix up the stack
          PLY

          PLB
          RTL

ActionTbl ANOP
          DC      I2'NoAction'
          DC      I2'NDA_Event'
          DC      I2'NDA_Run'
          DC      I2'NDA_Cursor'
          DC      I2'NDA_Rsvrd'
          DC      I2'NDA_Undo'
          DC      I2'NDA_Cut'
          DC      I2'NDA_Copy'
          DC      I2'NDA_Paste'
          DC      I2'NDA_Clear'

NoAction  ANOP
          RTS

NDA_Rsrvd ANOP
          RTS

NDA_Undo  ANOP
NDA_Cut   ANOP
NDA_Copy  ANOP
NDA_Paste ANOP
NDA_Clear ANOP

; On exit A=0 if edit command wasn't handled; non-zero if it was.
; You will usually want to say it was handled, because the application
; will not be active and so shouldn't be dealing with edit commands.

          LDA     #$FFFF             ;Say we handled it.
          RTS
```

```
; Display a wristwatch cursor if the cursor is over top of
; the content region of the window. This routine is only called
; when the DA window is the front window.

NDA_Cursor ANOP

           PHA
           PHA
           _GetPort                 ;Save current GrafPort

           PushLong WindowPtr
           _SetPort                 ;Make clock window active GrafPort

           PushPtr PortRect
           _GetPortRect             ;Get the port rectangle (local coords)

           PushPtr MousePosn        ;Return position in GrafPort coords

           _GetMouse                ;Get cursor position

           PHA                      ;space for result
           PushPtr MousePosn        ;pointer to mouse coordinate
           PushPtr PortRect         ;pointer to content region rectangle
           _PtInRect
           PLA                      ;Is point in content region?
           BEQ     NDA_Curs1        ;No, so branch

; Switch to watch cursor, but only if it's not already active:

           PHA
           PHA
           _GetCursorAdr            ;Get current cursor pointer
           PLA
           PLX

           CMP     #WatchCurs       ;Is it the watch?
           BNE     NDA_Curs0        ;Definitely not
           CPX     #^WatchCurs      ;Is it the watch?
           BEQ     NDA_Curs3        ;Yes, so do nothing

NDA_Curs0  PushPtr WatchCurs
           BRA     NDA_Curs2

; Switch to application cursor, but only if it's not already active:

NDA_Curs1  PHA
           PHA
           _GetCursorAdr            ;Get current cursor pointer
           PLA
           PLX
```

```
          CMP     #WatchCurs          ;Is it the watch?
          BNE     NDA_Curs3           ;No, so don't do anything
          CPX     #^WatchCurs         ;Is it the watch?
          BNE     NDA_Curs3           ;No, so don't do anything

          PushLong OldCursor          ;Switch to application cursor

NDA_Curs2 _SetCursor

NDA_Curs3 _SetPort                    ;Restore GrafPort
          RTS

; This subroutine is called once every "Period" ticks:

NDA_Run   ANOP

          PHA                         ;Space for result
          PHA
          _GetPort                    ;Save current port

          PushLong WindowPtr
          _SetPort                    ;Switch to clock window for drawing

          JSR ShowTime                ;Display the new time

          _SetPort                    ;(Pointer still on stack)

          RTS

; X and Y (pushed on stack) contain pointer to event record

NDA_Event ANOP

TheEvent  EQU     $05                 ;1 (base) + 2 (JSR) + 2 (PHD)

          PHD
          TSC
          TCD                         ;Align d.p. with stack

          LDA     [TheEvent]          ;Get "what" code
          CMP     #9                  ;Anything we support?
          BCS     TE1                 ;No, so branch

          ASL     A                   ;x2 to step into table
          TAX
          JSR     (EventTbl,X)

TE1       PLD                         ;Restore direct page
          RTS
```

```
EventTbl   ANOP
           DC       I2'NoEvent'       ;Not supported
           DC       I2'DoMouseDwn'    ; Mouse-down
           DC       I2'DoMouseUp'     ; Mouse-up
           DC       I2'DoKeyDwn'      ; Key down
           DC       I2'NoEvent'       ;Not supported
           DC       I2'DoAutoKey'     ; Autokey
           DC       I2'DoUpdate'      ; Update
           DC       I2'NoEvent'       ;Not supported
           DC       I2'DoActivate'    ; Activate

DoMouseUp  ANOP
DoMouseDwn ANOP
DoKeyDwn   ANOP
DoAutoKey  ANOP

NoEvent    RTS

DoUpdate   ANOP

           PushLong WindowPtr
           _BeginUpdate               ;Visible region = update region

           JSR      ShowTime          ;Display the current time

           PushLong WindowPtr
           _EndUpdate                 ;Restore entire visible region

           RTS

; If NDA window is deactivated, return to original cursor.

DoActivate ANOP

           LDY      #14               ;Access modifiers field
           LDA      [TheEvent],Y
           AND      #$01              ;Isolate activate/deactivate flag
           BEQ      NDA_Off           ;If 0, deactivate
           RTS

NDA_Off    PushLong OldCursor
           _SetCursor                 ;Switch to previous cursor
           RTS

ShowTime   ANOP

           PushPtr TheTime
           _ReadAsciiTime             ;Read the clock

           SEP      #$20              ;8-bit A register for byte accesses
           LONGA    OFF
```

```
          LDY     #19
ST1       LDA     TheTime,Y
          AND     #$7F              ;Convert to standard ASCII
          STA     TheTime,Y
          DEY
          BPL     ST1

          REP     #$20              ;Back to 16-bit A register
          LONGA   ON

          PushWord #20              ;horizontal
          PushWord #9               ;vertical
          _MoveTo

          PushPtr TheTime
          _DrawCString              ;Draw the time string

          RTS


; Startup or shutdown the NDA. On entry, A=0 for DeskShutDown,
; A is non-zero for DeskStartup.

NDA_Init  ANOP

          PHB
          PHK
          PLB

          CMP     #0                ;Starting up?
          BNE     NDA_Init1         ;Yes, so do nothing

          LDA     ClockOpen         ;Clock window open?
          BEQ     NDA_Init1         ;No, so branch

          PushLong WindowPtr
          _CloseWindow              ;Close the window (releases memory)

          STZ     ClockOpen         ;Set "closed" flag

NDA_Init1 PLB
          RTL

; The data area begins here:

NDA_Title STR 'Calendar/Clock'      ;Window title

WindowDef ANOP
          DC    I2'EndWind-WindowDef'
          DC    I2'%1100000010100000' ;Window with close box, title
          DC    I4'NDA_Title'       ;Pointer to window name
          DC     I4'0'
          DC     I2'0,0,0,0'
          DC     I4'0'
          DC     I4'0'                   ;Origin at (0,0)
          DC     I4'0'
          DC     I4'0'
          DC     I4'0'
          DC     I4'0'
          DC     I4'0'
          DC     I2'0'
          DC     I4'0'
          DC     I4'0'
          DC     I4'0'                   ;(Handle our own updates)
          DC     I'60,65,71,245'         ;Dimensions of window
          DC     I4'-1'                  ;Put clock window in front
          DC     I4'0'
EndWind   ANOP

WindowPtr DS       4                     ;Pointer to window record
ClockOpen DS       2                     ;Used as a flag

PortRect  DS       8                     ;Content region rectangle
MousePosn DS       4                     ;Current mouse position (local)

TheTime   DS       20                    ;ReadAsciiTime returns 20 bytes
          DC       C'        '           ;Add padding
          DC       I1'0'                 ;(terminator for DrawCString)

OldCursor DS       4                     ;Pointer to application's cursor recor

; This is the cursor record for a "wristwatch" cursor:

WatchCurs DC       I2'12'                ;Rows in cursor image
          DC       I2'3'                 ;Cursor width (in words)

          DC       H'000000000000'       ;The cursor image
          DC       H'000FF0000000'
          DC       H'000FF0000000'
          DC       H'00F00F000000'
          DC       H'0F00F0F00000'
          DC       H'0F00F0F00000'
          DC       H'0F0FF0FF0000'
          DC       H'0F0000F00000'
          DC       H'00F00F000000'
          DC       H'000FF0000000'
          DC       H'000FF0000000'
          DC       H'000000000000'

          DC       H'000FF0000000'       ;The cursor mask
          DC       H'00FFFF000000'
          DC       H'00FFFF000000'
DC      H'0FFFFFF00000'
DC      H'FFFFFFFF0000'
DC      H'FFFFFFFF0000'
DC      H'FFFFFFFFF000'
DC      H'FFFFFFFF0000'
DC      H'0FFFFFF00000'
DC      H'00FFFF000000'
DC      H'00FFFF000000'
DC      H'000FF0000000'

DC      I2'6,8'             ;Hot spot (y,x)

END
```

CHAPTER 10

The ProDOS 16 Operating System

The software interface between an application program and a mass-storage device such as a disk drive is called a *disk operating system*. Its main responsibilities are to organize data and program files on the disk medium and to provide a simple mechanism applications can use to transfer data to and from these files. Other common chores a disk operating system performs are file creation, deletion, and renaming, and volume management.

For the past few years, the standard operating system for the Apple II family has been ProDOS, the Professional Disk Operating System. The original release of ProDOS, since renamed ProDOS 8, was designed specifically for the IIc, IIe, and II Plus, and works in the 65C02's (or 6502's) 64K memory space only.

ProDOS 8 also works with a GS that is running IIe-style applications, of course, but it will not work with GS-specific applications, because they are not confined to the first 64K of memory. To solve this problem, Apple Computer, Inc. created ProDOS 16, an operating system that is similar in structure to ProDOS 8 but that takes advantage of the entire GS memory space.

ProDOS 16 runs in 65816 native mode, which means it works on the Apple IIGS only. If you try to boot it on another type of Apple II you will see an appropriate error message. Programs that work with ProDOS 8 will not work with ProDOS 16, because ProDOS 16 uses a new command calling sequence. Fortunately, programmers should find it relatively easy to adapt programs to ProDOS 16, because most ProDOS 8 commands have ProDOS 16 equivalents with the same symbolic names and similar types of command parameters.

Both versions of ProDOS format disks and store files in exactly the same way. Thus, data files can be accessed directly by either operating system. ProDOS 16 is even capable of switching to ProDOS 8 if the proper system files are included on the start-up disk.

This chapter looks at the main features of ProDOS 16 and gives several examples of how to use ProDOS 16 commands from within a program. It also reviews some of the jargon associated with ProDOS in general and ProDOS 16 in particular. ProDOS 8 has been covered in great detail in *Apple ProDOS: Advanced Features for Programmers* (Little, 1985). That subject matter will not be repeated here.

BLOCK-STRUCTURED DEVICES

ProDOS 16 works with *block-structured* storage devices only. A block is a group of 512 bytes of data and represents the smallest unit of information the device controller can read or write. Contrast this with *character* devices, such as printers or modems which deal with only one character at a time.

The most common block-structured devices used with ProDOS 16 on the GS are the following disk drives:

- 3 ½-inch disk drives (Apple 3.5 Drive, UniDisk 3.5)
- 5 ¼-inch disk drives (Apple 5.25 Drive, UniDisk 5.25, DuoDisk, Disk II)
- Hard disks (Hard Disk 20SC, ProFile)

Appendix 7 describes how to connect 3 ½-inch and 5 ¼-inch drives to the GS.

In addition, it is possible to partition and configure an area of memory so that ProDOS 16 thinks it is a block-structured device called a RAM disk. The advantage of using a RAM disk instead of a mechanical disk drive is that input/output operations are much quicker. You must remember to save files on the RAM disk to a real disk before you turn off the GS, however.

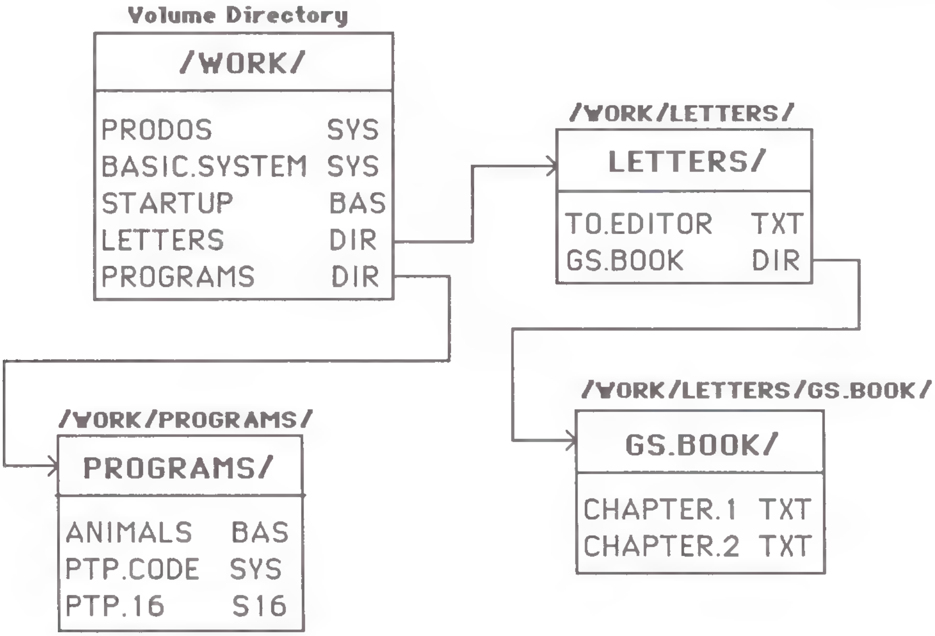
The easiest way to create a RAM disk is to use the Control Panel desk accessory. With it you can allocate all or a portion of the memory on a GS memory expansion card for RAM disk use. If you format this RAM disk and put the necessary operating system files on it, you can even boot from it when you press Control-OpenApple-Reset. To select it as the boot device, use the Control Panel Slots command to change the Boot option.

DIRECTORIES AND FILES

A *volume* is the general name for the storage medium used by a block-structured device. Each volume formatted by ProDOS 16 contains one or more *directories* in which files may be stored. The main directory, called the *volume directory* or the *root directory*, is created when you format the disk; it may contain up to 51 entries, representing ordinary data files or *subdirectory* files. A subdirectory may contain as many entries as space permits, including other subdirectories. See figure 10–1 for a diagram showing how multiple directories on a volume are interrelated.

The ability to create a hierarchy of directories makes it easy to manage large numbers of files on a single disk. This is because groups of related files can be

Figure 10–1. The ProDOS Hierarchical Directory Structure



isolated in their own separate directories where they will not be mixed with the many other files on the disk.

Every file on a disk is associated with a *filename* up to fifteen characters long. The first character must be a letter of the alphabet from A to Z, but subsequent characters can be letters, digits (0 to 9), or periods.

Because of the way directories can be nested when using ProDOS 16, it is not enough to identify a file by its name only. You must also identify the directory in which the file is stored. The identifying string required by ProDOS 16 is called a *pathname*. It is a concatenation of the names of each of the directories ProDOS 16 must pass through to reach the file's subdirectory, followed by the name of the file itself. The pathname begins with a slash (/) and each directory name in the pathname is separated from the next with a slash.

Suppose the name of the volume directory of your disk is WORK and within this directory you have defined a subdirectory called LETTERS that contains a file called TO.EDITOR. The pathname to use to identify this file is:

```
/WORK/LETTERS/TO.EDITOR
```

You can avoid specifying a complete pathname every time you want to deal with a file by setting a *default prefix*. A default prefix identifies the path to a particular directory, and ProDOS 16 automatically puts that path in front of any filename you specify. It will also put it in front of a *partial pathname*. A partial pathname is a name describing a series of directories that does not begin with a slash. Essentially, a partial pathname is the pathname of a file relative to the directory described by the default prefix.

For example, if you set the default prefix to /WORK/LETTERS in the above example, you could identify TO.EDITOR by its filename only. If the default prefix was /WORK/, you would specify the partial pathname LETTERS/TO.EDITOR.

The default prefix can also be represented by the name 0/. Thus, 0/TO.EDITOR is equivalent to /WORK/LETTERS/TO.EDITOR if the 0/ prefix has been set to /WORK/LETTERS. When an application first starts up, the default prefix identifies the name of the boot volume.

As indicated in table 10–1, ProDOS 16 has a similar shorthand notation for the prefixes describing eight other directories. Only 0/ and 3/ to 7/ should be changed by an application, although it is possible to change 1/ and 2/ if you really need to. Change prefixes with the SET_PREFIX command.

THE STRUCTURE OF A PRODOS 16 BOOT DISK

Certain files must be present on a ProDOS 16 system disk before you can boot it or use it to run both ProDOS 8 and ProDOS 16 applications. The structure of the simplest such system disk is as follows:

```
PRODOS                 Operating system loader
SYSTEM/                Subdirectory: operating system files
  P8                   The ProDOS 8 operating system
  P16                  The ProDOS 16 operating system
  START                The start-up program
  TOOLS/               Subdirectory: RAM-based tool sets
  FONTS/               Subdirectory: font files
  DESK.ACCS/           Subdirectory: desk accessories
  LIBS/                Subdirectory: system library files
  DRIVERS/             Subdirectory: device drivers
  SYSTEM.SETUP/        Subdirectory: initialization programs
    TOOL.SETUP           Patches to ROM-based tool sets
```

A ProDOS 16 system disk goes through a rather convoluted start-up procedure when you boot it. It begins by loading the ProDOS program into memory and executing it. Once ProDOS gets control it loads the ProDOS 16 operating system from the SYSTEM/P16 file and then executes the TOOL.SETUP program in the SYSTEM/SYSTEM.SETUP/ directory. TOOL.SETUP patches and enhances ROM-based tool sets.

Table 10–1: Standard Prefix Numbers

Prefix Number	*Description*
*/	The boot prefix This is the volume name from which ProDOS 16 was booted. This prefix cannot be changed.
0/	The default prefix ProDOS automatically attaches it to any filename or partial pathname (as opposed to full pathname) you specify.
1/	The application prefix The pathname of the directory containing the current application program.
2/	The system library prefix The pathname of the directory containing library modules used by the current application. For a standard ProDOS 16 boot disk, this is /MYDISK/SYSTEM/LIBS/.
3/	User-definable
4/	User-definable
5/	User-definable
6/	User-definable
7/	User-definable

ProDOS then continues by loading and executing every other file in the SYSTEM/SYSTEM.SETUP/ directory. Typical files include permanent initialization (start-up) files with file type codes of $B6 and temporary initialization files with file type codes of $B7. The difference between these two types of files is that temporary initialization files remove themselves from memory when they finish executing; permanent initialization files do not.

ProDOS then moves to the SYSTEM/DESK.ACCS/ directory and loads into memory any Classic Desk Accessory files (file type $B9) and New Desk Accessory files (file type $B8). This causes the names of the Classic Desk Accessories to be placed in the menu that appears when you press Control-OpenApple-Esc.

Next, ProDOS searches the SYSTEM/ directory for a file called START that has a file type of $B3 (S16). If it finds this file, it loads and executes it and the boot process ends. The START program is typically a program selector that lets you choose a particular application to run.

If PRODOS does not find START, it scans the volume directory until it finds a ProDOS 8 system program (file type $FF) whose name ends with ".SYSTEM" or a ProDOS 16 system program (file type $B3) whose name ends with ".SYS16." It then ends the boot procedure by running the program. It will not run a ProDOS 8 program unless SYSTEM/P8 is on the disk, however. If no such system program is found, PRODOS brings up a window asking the user to enter the pathname of the application to run.

USING PRODOS 16 COMMANDS

The general procedure for calling a ProDOS 16 command is different from the one used to call a tool set function. It goes something like this:

```
    JSL    $E100A8              ;Call ProDOS 16 entry point
    DC     I2'CommandNum'       ;Command number [word]
    DC     I4'ParmTable'        ;Address of parameter table [long]
    BCS    Error                ;(Control resumes here after call)
```

You can call a ProDOS 16 command while the 65816 is in native mode.

Immediately following the JSL instruction is a word containing the identification number of the ProDOS 16 command you wish to use. Table 10–2 contains a list of all 32 ProDOS 16 commands and command numbers.

Following the command number is a pointer to a parameter table containing parameters required by the command and results returned by the command. The parameters can be one- or two-word numeric values or two-word pointers. The exact structure of the parameter table varies from command to command.

When a ProDOS 16 command finishes, it adds 6 to the return address pushed on the stack by the JSL instruction, then ends with an RTL instruction. This causes control to pass to the code beginning just after the pointer to the parameter table. On return, all registers remain unchanged except the accumulator (which contains an error code), the program counter, and the status register (the m, x, I, and e flags are unchanged; N, V, and Z are undefined; the D flag is cleared; the carry flag reflects the error status).

At this stage, you can check the state of the carry flag to determine whether an error occurred: if the carry flag is clear, there was no error; if it is not clear, an error did occur. An error code indicating the nature of the error comes back in the accumulator; the accumulator will contain 0 if no error occurred. See table R10–2 in the reference section at the end of this chapter for a list of ProDOS 16 error codes.

APW comes with a set of macros which will help to simplify the use of a ProDOS 16 command. They are stored in a file called M16.PRODOS. To call a ProDOS command with a macro, use instructions of the form:

```
    _CMDNAME ParmTbl
```

Table 10–2: The ProDOS 16 Commands

Command Number	*Command Name*	*Description of Command*
$01	CREATE	Creates a new file
$02	DESTROY	Deletes a file
$04	CHANGE_PATH	Renames a file or moves it to another directory
$05	SET_FILE_INFO	Changes the attributes of a file
$06	GET_FILE_INFO	Returns the attributes of a file
$08	VOLUME	Returns the name and attributes of the volume in a device
$09	SET_PREFIX	Assigns a pathname prefix to one of the eight standard prefix numbers
$0A	GET_PREFIX	Returns the pathname prefix for one of the eight standard prefix numbers
$0B	CLEAR_BACKUP_BIT	Clears the “backup-needed” bit in a file’s access code byte
$10	OPEN	Prepares a file for subsequent read, write, and positioning commands
$11	NEWLINE	Sets the end-of-line character for read operations
$12	READ	Reads data from an open file
$13	WRITE	Writes data to an open file
$14	CLOSE	Prevents access to a file until it is reopened
$15	FLUSH	Writes the contents of the file’s I/O buffer to disk
$16	SET_MARK	Sets the active position in an open file
$17	GET_MARK	Returns the active position in an open file
$18	SET_EOF	Sets the size of the file
$19	GET_EOF	Returns the size of the file

Table 10–2: Continued

Command Number	*Command Name*	*Description of Command*
$1A	SET_LEVEL	Sets the system file level
$1B	GET_LEVEL	Returns the system file level
$20	GET_DEV_NUM	Returns the number of a named device
$21	GET_LAST_DEV	Returns the number of the last device accessed
$22	READ_BLOCK	Reads a block of data from a device
$23	WRITE_BLOCK	Writes a block of data to a device
$24	FORMAT	Formats the medium in a device
$27	GET_NAME	Returns the filename of the current application
$28	GET_BOOT_VOL	Returns the name of the ProDOS boot volume
$29	QUIT	Exits the current application
$2A	GET_VERSION	Returns the ProDOS version number
$31	ALLOC_INTERRUPT	Installs an interrupt handler
$32	DEALLOC_INTERRUPT	Removes an interrupt handler

where CMDNAME represents the name of the command and ParmTbl represents the address of the parameter table associated with the command. At assembly time, this macro is expanded into the standard ProDOS 16 calling sequence.

The main advantage of using the macros is you do not have to memorize command numbers, only command names. It also makes assembly language programs that use ProDOS 16 a lot easier to read.

In the sections which follow, parameter tables for ProDOS 16 commands are presented. These tables describe the order of the parameters, the size of the parameters, and whether the parameter is an Input (I) or a Result (R). An Input is a parameter that must be provided before using the command. A Result (R) is a parameter returned by the command.

Note that even though a pointer to a string may be marked as a result, ProDOS 16 does not actually return the pointer. Instead, it returns the bytes in the string in the space pointed to by the pointer. It is the responsibility of the application to allocate a space of the proper size and to provide a pointer to it.

FILE-MANAGEMENT COMMANDS

The ProDOS file-management commands perform a variety of important operations on closed files:

- Creating, destroying, and renaming files
- Reading and changing file attributes
- Reading and changing pathname prefixes

The first two groups of commands affect only the directory entry for a file, not the data inside the file.

Creating New Files

The file management command you will probably use most often is CREATE. With it you can create a directory entry for a data file; you can subsequently open this file and write data to it. You can also use CREATE to create subdirectory files.

The parameter list for CREATE is quite lengthy, because it must contain all the attributes for the file. Here is how it is structured:

CREATE ($01)

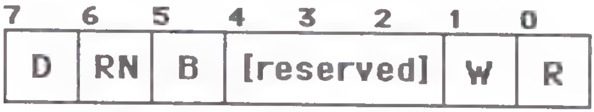
Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string
+4 to +5	access	I	Access code
+6 to +7	file_type	I	File type code
+8 to +11	aux_type	I	Auxiliary type code
+12 to +13	storage_type	I	Storage type code
+14 to +15	create_date	I	Creation date
+16 to +17	create_time	I	Creation time

Every item in the CREATE parameter list is an input parameter, so each must be set up properly before calling CREATE. Multibyte items are stored with the low-order bytes first, as usual.

There are many unfamiliar items in this parameter list that require further explanation; the same items appear in the parameter tables of many other ProDOS 16 commands. Each item is examined in detail below.

Pathname. This item is the address of a pathname string identifying the file to be created. The string begins with a length byte and is followed by ASCII-encoded characters.

Access. The access item indicates whether the file may be read from, written to, renamed, or destroyed; it also indicates whether the file has been modified since the last back-up operation. Only the low-order byte of access is defined (the high-order byte is 0); the meaning of each bit is as follows:



7	6	5	4	3	2	1	0
D	RN	B	[reserved]			W	R

The abbreviations in this chart have the following.meanings:

- D = delete-enabled
- RN = rename-enabled
- B = back-up required
- W = write-enabled
- R = read-enabled

The access attribute for a particular bit is enabled if the corresponding bit is 1; if the bit is 0, the attribute is disabled.

When creating standard files, with full access permitted, set access to $E3. If some of the access attributes are disabled, the file is said to be locked.

ProDOS 16 automatically sets the back-up bit to 1 whenever you write to the file to inform a back-up utility program that the file has changed since the last back-up. It is the responsibility of the back-up to clear this bit to 0 with the CLEAR_BACKUP_BIT command after making the back-up.

File_type. The low-order byte of the file type code indicates the general nature of the data the file contains. (The high-order byte is always 0.) A list of the commonly-used codes is provided in table 10–3.

Table 10-3: ProDOS 16 File Type Codes

File Type Code	*Standard Mnemonic*	*Description of File*
$00		Uncategorized file
$01	BAD	Bad block file
$02	PCD	Pascal code (SOS)
$03	PTX	Pascal text (SOS)
$04	TXT	ASCII textfile
$05	PDA	Pascal data (SOS)
$06	BIN	General binary file
$07	FNT	Font file (SOS)
$08	FOT	Graphics screen file
$09	BA3	Business BASIC program (SOS)
$0A	DA3	Business BASIC data (SOS)
$0B	WPF	Word processor file (SOS)
$0C	SOS	SOS system file
$0D–$0E		[Reserved for SOS]
$0F	DIR	Subdirectory file
$10	RPD	Record Processing System data (SOS)
$11	RPI	Record Processing System index (SOS)
$12		AppleFile discard file (SOS)
$13		AppleFile model file (SOS)
$14		AppleFile report format file (SOS)
$15		Screen library file (SOS)
$16–$18		[Reserved for SOS]
$19	ADB	AppleWorks database file
$1A	AWP	AppleWorks word processing file
$1B	ASP	AppleWorks spreadsheet file
$1C–$AF		[Reserved]
$B0	SRC	APW source code

Table 10–3: Continued

File Type Code	*Standard Mnemonic*	*Description of File*
$B1	OBJ	APW object code
$B2	LIB	APW library
$B3	S16	ProDOS 16 system program
$B4	RTL	APW run-time library
$B5	EXE	APW executable shell application
$B6	STR	ProDOS 16 permanent init (start-up) file
$B7	TSF	ProDOS 16 temporary init file
$B8	NDA	New Desk Accessory
$B9	CDA	Classic Desk Accessory
$BA	TOL	Tool set
$BB	DRV	ProDOS 16 device driver
$BC–$BE		[Reserved for ProDOS 16 load files]
$BF	DOC	ProDOS 16 document file
$C0	PNT	Compressed super high-res picture file
$C1	PIC	Super high-res picture file
$C2–$C7		[Reserved]
$C8	FON	ProDOS 16 font file
$C9–$EE		[Reserved]
$EF	PAS	Pascal area on a partitioned disk
$F0	CMD	ProDOS 8 added command file
$F1–$F8		ProDOS 8 user-defined files
$F9	P16	ProDOS 16 file
$FA	INT	Integer BASIC program
$FB	IVR	Integer BASIC variables
$FC	BAS	Applesoft BASIC program
$FD	VAR	Applesoft BASIC variables
$FE	REL	EDASM relocatable code file
$FF	SYS	ProDOS 8 system program

NOTE: SOS stands for the Apple III Sophisticated Operating System.

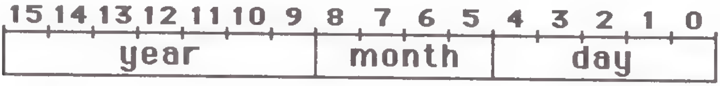
Aux_type. The meaning of the auxiliary type code for a file depends on the file type code. A textfile, for example, uses the auxiliary type code to indicate the random-access record length (or 0 for a sequential textfile not divided into separate records). APW uses the aux_type code in SRC ($B0) files to hold a language ID number. Note that only the low-order word of aux_type is presently used by ProDOS 16.

Storage_type. The low-order byte of the storage type code indicates the structure of the file on the disk. The possible values are:

$00	Inactive entry
$01	Seedling file (EOF <= 512 bytes)
$02	Sapling file (512 < EOF <= 128K bytes)
$03	Tree file (128K < EOF < 16M bytes)
$04	Pascal region on a partitioned disk
$0D	Subdirectory file
$0E	A subdirectory header
$0F	A volume directory header

The differences among a seedling, a sapling, and a tree file are really not important to the designer of an application program. Suffice to say that all standard data files should have a storage type of $01 (seedling); as the file grows in size, ProDOS 16 automatically changes its structure into that of a sapling file and then into a tree file. For a description of these three basic file structures, refer to *Apple ProDOS: Advanced Features for Programmers* (Little, 1985). If you are creating a subdirectory file, set storage_type to $0D.

Create_date. The creation date is a two-byte value containing the year, month, and day on which the file was created. The encoded formats of these three quantities are as follows:



If you specify a creation date of 0, ProDOS 16 automatically uses the current date.

Create_time. The creation time is a two-byte value containing the hour and minute on which the file was created. These two quantities are encoded as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	hour					0	0	minute					

If you specify a creation time of 0, ProDOS 16 automatically uses the current time.

Deleting Files

To remove a file from its directory permanently, you must delete the file with the DESTROY command. Destroying a file also frees up the disk blocks used by the file, making them available for allocation to other files.

The only item in the DESTROY parameter list is a pathname pointer:

DESTROY ($02)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string

Note that you cannot destroy a file if the delete-enabled bit in its access code is 0. You can use SET_FILE_INFO (see below) to set this bit prior to destroying a file.

Renaming Files

The most common use for the CHANGE_PATH command is to rename a file, but it is also useful for moving a file from one subdirectory to another on the same disk. Its parameter table looks like this:

CHANGE_PATH ($04)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string
+4 to +7	new_pathname	I	Pointer to the new pathname string

When the two pathnames specified are identical except for the filename portion, CHANGE_PATH simply renames the file. If the pathnames describe files in different directories, the entry for the file in the first directory specified is moved to the second.

Note that you cannot rename a file if the rename-enabled bit in its access code is 0. You can use SET_FILE_INFO (see below) to set this bit prior to renaming a file.

Changing File Attributes

As was discussed in the examination of the CREATE command, each file is associated with a set of attributes that describes such things as the date and time on which the file was created, the file type, the auxiliary type code, and so on. You can determine what these parameters are for any existing file using the GET_FILE_INFO command. Here is what its parameter table looks like:

GET_FILE_INFO ($06)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string
+4 to +5	access	R	Access code
+6 to +7	file_type	R	File type code
+8 to +11	aux_type	R	Auxiliary type code[1]
+12 to +13	storage_type	R	Storage type code
+14 to +15	create_date	R	Creation date
+16 to +17	create_time	R	Creation time
+18 to +19	mod_date	R	Modification date
+20 to +21	mod_time	R	Modification time
+22 to +25	blocks_used	R	Blocks used by the file

[1] For a volume directory file, this field becomes total_blocks (the number of blocks on the volume).

The only input parameter is the pathname describing the file you want to examine. All other parameters are returned by GET_FILE_INFO.

The last three types of parameters in the chart above have not been discussed yet:

> *Mod_date.* The date on which the file was last modified. The format of this word is the same as that for create_date.
>
> *Mod_time.* The time at which the file was last modified. The format of this word is the same as that for create_time.
>
> *Blocks_used.* The total number of blocks used by the file. It includes any non-data blocks used as overhead by the operating system.

The results returned by GET_FILE_INFO are slightly different if the pathname points to the name of the volume directory of the disk. In this case, the aux_type field contains the size of the disk in blocks (called *total blocks*) and the blocks_used field contains the total number of blocks used by all files on the disk.

You can use a related command, SET_FILE_INFO, to change the attributes of any file; its parameter table looks just like the one used for GET_FILE_INFO, except that it does not include the blocks_used field at the end. An easy way to change one particular file attribute without affecting the rest is to call GET_FILE_INFO, store the new parameter in the GET_FILE_INFO parameter table, and then call SET_FILE_INFO using the same parameter table. Note, however, that you cannot change the storage_type attribute with SET_FILE_INFO.

Note that SET_FILE_INFO cannot be used to clear the back-up-needed bit of the access code word. This bit is primarily for the benefit of disk-back-up utilities. By examining the bit, such utilities can determine if a file has changed since the last back-up. After the back-up operation, the back-up utility should clear the back-up bit to 0. To do this, it must use the CLEAR_BACKUP_BIT command. Here is the structure of the parameter table for CLEAR_BACKUP_BIT:

CLEAR_BACKUP_BIT ($0B)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string

Notice that all you have to specify is a pointer to the pathname string.

Determining Volume Characteristics

The VOLUME can be used to determine some of the characteristics of a disk volume:

VOLUME ($08)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	dev_name	I	Pointer to the device name string
+4 to +7	vol_name	R	Pointer to the volume name string
+8 to +11	total_blocks	R	Size of the volume in blocks
+12 to +15	free_blocks	R	Number of unused blocks
+16 to +17	file_sys_id	R	Operating system ID code

The VOLUME command takes one input parameter—a pointer to the name of the device (dev_name)—and returns information respecting the volume in that device.

Dev_name points to a string of the form .Dn that is preceded by a length byte. The value for n can be from 1 up to the number of ProDOS block devices in the system. The only way to determine the maximum value for n is to call VOLUME with consecutive device names until error $11 (invalid device) occurs.

Vol_name is a 17-byte buffer area in which VOLUME returns a length byte followed by the name of the volume. The name of the volume includes a leading slash (/).

The other values returned are as follows:

total_blocks. The total number of blocks on the volume.

free_blocks. The total number of unused blocks on the volume.

file_sys_id. The low-order byte of this word indicates the type of operating system to which the device belongs. The possible results are as follows:

- 1 = ProDOS or SOS
- 2 = DOS 3.3
- 3 = DOS 3.1 or 3.2
- 4 = Apple II Pascal
- 5 = Macintosh (flat file structure)

- 6 = Macintosh (HFS: hierarchical file structure)
- 7 = Macintosh XL
- 8 = Apple CP/M

All other values are reserved. Version 1.1 of ProDOS 16 recognizes only ProDOS and SOS disks, however (file_sys_id = 1). If it detects another type of disk, it returns an error code of $52 (unsupported volume type).

The other common error codes that VOLUME might return are as follows:

$27	I/O error; this error is reported if there is no disk in a 5¼-inch drive.
$28	No device connected; this error occurs if you do not have a second 5¼-inch drive connected to the drive controller.
$2F	Device not on line; this occurs if there is no disk in a 3½-inch drive.

Manipulating Prefixes

Earlier in this chapter you saw how useful prefixes can be. ProDOS 16 lets you define eight different prefixes, which can be referred to by the prefix designators 0/, 1/, 2/, 3/, 4/, 5/, 6/, and 7/. As explained earlier, the first three prefixes are reserved for the system subdirectory (0/), the application subdirectory (1/), and the library subdirectory (2/).

The prefixes from 3/ to 7/ are not used in any special way by the operating system, so applications can use them to identify any directories they wish. For example, 4/ could be a directory containing help files and 5/ could be a directory containing data files for an application.

Use the SET_PREFIX command to assign prefix strings to the eight standard prefixes. The parameter table looks like this:

SET_PREFIX ($09)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	prefix_num	I	Prefix number (0 to 7)
+2 to +5	prefix	I	Pointer to the new prefix string

Both these parameters are input parameters. The string pointed to by prefix begins with a length byte, which is followed by the ASCII-encoded characters in the directory pathname. If the prefix string is not preceded by a slash, the string is appended to the current prefix to create the new prefix string.

The GET_PREFIX parameter table looks like this:

GET_PREFIX ($0A)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	prefix_num	I	Prefix number (0 to 7)
+2 to +5	prefix	R	Pointer to prefix name string

Use GET_PREFIX to determine the current settings of the standard prefixes. It returns the result in a buffer pointed to by the prefix parameter. The result is a length byte followed by the prefix name. The name is preceded and followed by slashes ("/"). You must allocate a buffer of 67 bytes to accommodate the largest prefix name that might be returned.

FILE I/O COMMANDS

The file I/O commands affect the data portion of a file. The two main commands an application uses are for reading and writing. Reading, of course, is the transfer of data from disk to memory; writing involves a transfer in the opposite direction.

Positions in the File

Before the file I/O commands are examined, two important concepts—*mark* and *EOF*—should be mentioned.

The position within a file at which a subsequent read or write operation will take place is called the *mark position*. As you access bytes in the file, the mark position keeps incrementing, so it is always pointing to the next byte to be accessed.

When you open a file prior to reading from it or writing to it, ProDOS 16 sets mark to 0, meaning that subsequent operations will take place at the beginning of the file. (The bytes in a file are numbered from 0.) As will be shown later in this chapter, it is easy to skip to any other position in the file using the ProDOS 16 SET_MARK command.

The other important position parameter is EOF. EOF, which stands for "end-of-file," contains the size of the file in bytes, so it can be thought of as a pointer to the byte past the last byte in the file. EOF automatically increases as you write data to the end of the file. You can increase or decrease EOF explicitly with the SET_EOF command.

Reading

To read data from a file, you must perform the following steps:

1. Open the file
2. Move the mark position (if necessary)
3. Adjust newline mode (if necessary)
4. Read the data
5. Close the file

You can read only from a file which is open. Opening a file tells ProDOS 16 the name of the file you want to deal with and permits it to set up a buffer area it needs to manage the flow of data to and from the file. To open a file, use the OPEN command; it requires the following parameter table:

OPEN ($10)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	R	Reference number for the file
+2 to +5	pathname	I	Pointer to the pathname string
+6 to +9	null_field	R	Reserved area for ProDOS 16

The only parameter you specify before calling OPEN is the pathname pointer. The parameter OPEN returns is ref_num. It is an identifying number for the open file; this number is used instead of a pathname by all other ProDOS 16 commands dealing with open files. You must store ref_num in the reference number fields of the parameter tables for any other such ProDOS 16 commands you will call before closing the file.

ProDOS 16 sets the mark position to 0 when you open a file—the start of the file. Before actually reading data from the file, you may want to move to some other position. For example, if the file is composed of a series of 50-byte records and you want to move directly to record #3 (numbering begins with #0), you would move mark to position 150. This is done with the SET_MARK command; its parameter table looks like this:

SET_MARK ($16)

Offset	Symbolic Name	Input or Result	Description
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	position	I	The new mark position

The ref_num in the SET_MARK parameter table must be the same as the one returned when you opened the file whose mark position is being changed.

Another matter you might want to deal with before reading is to adjust the *newline* character. This is the character that, when read from a file, will terminate a read operation even if the requested number of characters has not been read. If you assign the newline character to a carriage return (ASCII $0D), for example, you can easily read a text file line by line.

The command for setting the newline character is NEWLINE. Its parameter table looks like this:

NEWLINE ($11)

Offset	Symbolic Name	Input or Result	Description
+0 to +1	ref_num	I	Reference number for the file
+2 to +3	enable_mask	I	Newline enable mask
+4 to +5	newline_char	I	Newline character

A character read from an open file is logically ANDed with the value of enable_mask before a comparison with newline_char is made. If the end-of-line character might be a carriage return whose high bit is on or off, use an enable_mask of $7F and a newline_char of $0D; when you do so, a read will terminate with $0D or $8D.

Set the enable_mask to $00 to disable the newline feature.

You are now ready to read information from the file with the READ command. The parameter table looks like this:

READ ($12)

Offset	Symbolic Name	Input or Result	Description
+0 to +1	ref_num	I	Reference number for file
+2 to +5	data_buffer	I	Pointer to start of data buffer
+6 to +9	request_count	I	Number of bytes to read
+10 to +13	transfer_count	R	Number of bytes actually read

You must specify three types of information in the parameter list: the reference number of the file to be read, the starting address of a block of memory where the input data will be stored, and the number of bytes to be read. When the operation ends, the transfer_count field contains the number of bytes actually read from the file. This number will be less than the requested number if the active newline character was encountered or if the end of the file was reached. If no characters were read at all, the carry flag is set and the accumulator contains an error code of $4C (end-of-file encountered).

As data is read from a file, the mark pointer automatically advances, so there is no need to call SET_MARK explicitly after each read.

When you are all through with a file you should formally close it with the CLOSE command. This frees up the memory spaces that ProDOS 16 allocates to the file when it is first opened. The parameter table for the CLOSE command looks like this:

CLOSE ($14)

Offset	Symbolic Name	Input or Result	Description
+0 to +1	ref_num	I	Reference number for the file

Once a file is closed, you cannot read from it again until it has been reopened.

If the ref_num specified in a CLOSE command is $00, all files at or above the current file level are closed. The file level is a number between 0 and 255 that you can set using the SET_LEVEL command:

SET_LEVEL ($1A)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	level	I	The new file level

To assign a certain file level to an open file, use SET_LEVEL before opening the file with OPEN. You cannot change the level of a file after it is open.

If you want to determine the current file level, use GET_LEVEL. Its parameter table is the same as that for SET_LEVEL.

You will not need to bother with file levels often. They become important when you want to manage EXEC files properly. EXEC files are files to which the operating system looks for input instead of the keyboard. If you switch to a higher file level after opening an EXEC file, subsequent CLOSE operations will not close the EXEC file.

The subroutine in listing 10–1 shows how to use many of the ProDOS 16 commands just described. It opens a textfile, disables newline mode, determines the file size with GET_EOF, uses the Memory Manager to allocate a block of that size, reads the file into memory, and then closes the file.

Writing to a File

The procedure for writing to a file (see listing 10–2) is similar to the one for reading from a file. Before actually writing, you must open the file and, if necessary, position the mark pointer. To perform the write operation, use the WRITE command; its parameter table looks like this:

WRITE ($13)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	data_buffer	I	Pointer to start of data buffer
+6 to +9	request_count	I	Number of bytes to write
+10 to +13	transfer_count	R	Number of bytes actually written

The data_buffer field points to a block of data you wish to write to the file; the size of this block is given by request_count. The transfer_count field returns the actual number of bytes written to the disk. It will be less than the requested number if the disk became full during the write operation or if a disk error occurred.

One common form of write operation is an *append*. This is the process of expanding the size of a file by adding data to it. The following subroutine shows how to append data to an existing file:

```
             _OPEN OpenParms          ;Open the file
             LDA ref_num              ;Get reference number
             STA ref_num1             ; and store it in other
             STA ref_num2             ; parameter lists.
             STA ref_num3             ;
             _GET_EOF EOFParms        ;Get the file size
             _SET_MARK EOFParms       ;Set mark to end of file
             _WRITE WriteParms        ;Write data to file
             _CLOSE CloseParms        ;Close the file
             RTS

OpenParms    ANOP
ref_num      DS   2                   ;Reference number
             DC   I4'FileName'        ;Pointer to pathname
             DS   4                   ;Handle to I/O buffer

EOFParms     ANOP
ref_num1     DS   2
position     DS   4                   ;Position in file

WriteParms   ANOP
ref_num2     DS   2
data_buff    DC   I4'Buffer'          ;Buffer containing the data
request      DC   I4'512'             ;Number of bytes to write
transfer     DS   4

CloseParms   ANOP
ref_num3     DS   2

FileName     STR 'MYFILE.DEMO'        ;Name of file

Buffer       DS   512                 ;Data to write
```

This example uses one command that has not been discussed yet: GET_EOF. GET_EOF returns the size of the file, in bytes, in a four-byte field in its parameter table. To do an append, all you must do is set the mark position to this value before writing to the file, as shown in the example above.

The complete parameter table for GET_EOF looks like this:

GET_EOF ($19)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	eof	R	The end-of-file position

Two other commands you may use in conjunction with write operations are GET_MARK and SET_EOF. Here are their parameter tables:

GET_MARK ($17)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	position	R	The current mark position

SET_EOF ($18)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	eof	I	The new end-of-file position

GET_MARK returns the current value of the mark pointer and SET_EOF sets the size of the file to the value passed in its parameter table. You will use both of them in situations in which you want to eliminate everything in a file past the current mark position. Here is the subroutine to use:

```
    _GET_MARK MarkParms         ;Get current mark position
    _SET_EOF MarkParms          ; and make it the new file size.
    RTS
```

Such an operation is often performed in word processing programs when a file is loaded into memory, modified, and saved to disk under the same name. If the modified file is shorter than the original, and you do not set the new EOF position after the write, the file will still include the tail end of the original file.

When you write data to a file, the data actually is not saved to disk right away. Instead, it is placed in a ProDOS I/O buffer that is transferred to disk only when it fills up or when the file is closed. You can force ProDOS to empty the I/O buffer at any time using the FLUSH command:

FLUSH ($15)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file

Calling FLUSH minimizes the risk of losing data if the system crashes or the power goes off before the file you are writing to is closed. It does slow down the effective execution speed of a program, however, so it is not used often.

Note that if ref_num is $00, all open files are flushed.

DEVICE-MANAGEMENT COMMANDS

The device-management commands are for dealing with block devices themselves, not the individual files they contain. They perform such chores as low-level block I/O and disk formatting.

Formatting

Disks must be formatted with the FORMAT command before you can save files to them. Formatting is actually a two-step process:

1. Conditioning the magnetic surface of the disk
2. Writing boot code, directory, and block usage information on the disk in the manner dictated by the operating system

Because formatting permanently erases any information on a disk, you should ask the user to verify such an operation if the disk is not blank.

The structure of the parameter table for FORMAT is as follows:

FORMAT ($24)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	dev_name	I	Pointer to the device name string
+4 to +7	vol_name	I	Pointer to the volume name string
+8 to +9	file_sys_id	I	Operating system ID code

The dev_name parameter is a pointer to the device name string (the string will be .D1, .D2, and so on). The volume name string can be up to 16 characters long; it must adhere to the ProDOS file-naming rules and must include a leading slash (/).

The meaning of the file_sys_id field was explained earlier in connection with the VOLUME command. It describes the operating system used by the command. In this case, it tells FORMAT the technique to use to lay out the disk directory information on the volume.

Keep in mind that early versions of ProDOS 16 supported the formatting of ProDOS disks only (file_sys_id = $01). You will get an error code of $5D (operating system not supported) if you set file_sys_id to anything other than $01.

Accessing Specific Blocks

If you are writing a utility program such as a disk copier or a file undeleter, you will probably want to read and write specific blocks on a disk. To do this, use the READ_BLOCK and WRITE_BLOCK commands. They both use the same type of parameter block:

READ_BLOCK ($22)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	dev_num	I	Device reference number
+2 to +5	data_buffer	I	Pointer to the read buffer
+6 to +9	block_num	I	Block number to read in

WRITE_BLOCK ($23)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	dev_num	I	Device reference number
+2 to +5	data_buffer	I	Pointer to write buffer
+6 to +9	block_num	I	Block number to be written to

The memory blocks pointed to by data_buffer in each case are exactly 512 bytes long (the size of a disk block). The block numbers on a disk range from 0 to an upper limit that depends on the nature of the disk device. Use the VOLUME command to determine the size of a disk volume in blocks.

The dev_num parameter needed by both READ_BLOCK and WRITE_BLOCK is a device reference number (.D1, .D2, and so on), not the device name. To determine the reference number for a given device name, use the GET_DEV_NUM command; its parameter table is as follows:

GET_DEV_NUM ($20)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	dev_name	I	Pointer to the device name string
+4 to +5	dev_num	R	Device reference number

When ProDOS first starts up, it scans the system for block devices and assigns them unique device reference numbers. These numbers are consecutive integers beginning with 1. This identification system is quite different from that used in ProDOS 8 in which devices are identified by slot/drive combinations.

Last Device Accessed

ProDOS 16 has a command for determining the reference number of the last device accessed by a read or write command. This is the GET_LAST_DEV command:

GET_LAST_DEV ($21)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	dev_num	R	Device reference number

If GET_LAST_DEV is unable to determine what the last device was, it returns an error code of $60.

OPERATING-ENVIRONMENT COMMANDS

You can use the operating environment commands to determine the name of the currently running application, the name of the boot volume, and the ProDOS 16 version number. These commands also include the important QUIT command, which you will use to transfer control from one application to another.

Status Commands

It is often convenient for a program to know its own name. It may need to know this information so it can transfer a copy of itself to a RAM disk, for example. Instead of using a specific name, you should use the GET_NAME command, just in case the user has renamed the program.

The parameter table for GET_NAME looks like this:

GET_NAME ($27)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	data_buffer	R	Pointer to application name string

The space for the name should be 16 bytes long so that there will be room for the longest possible filename and the preceding length byte.

Notice that GET_NAME returns the filename only. The subdirectory in which it resides is given by the 1/ prefix. Use GET_PREFIX to determine what this prefix is.

To access files on the boot disk, you can specify a pathname constructed by adding the "*/" prefix designator to a partial pathname. If you need to know the actual name of the boot disk, use the GET_BOOT_VOL command. Here is its parameter table:

GET_BOOT_VOL ($28)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	data_buffer	R	Pointer to the volume name string

GET_BOOT_VOL returns a name which begins and ends with a slash, so the data_buffer space should be 18 bytes long.

There is also a GET_VERSION command for determining the version number of ProDOS. An application should check the version number if the program works only with certain versions of ProDOS. Here is the parameter table:

GET_VERSION ($2A)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	version	R	ProDOS 16 version number

The low-order byte of the version word represents the minor release number and the high-order byte represents the major release version. Version 2.1, for example, would be stored as $0201. If the high-order bit of the result is 1, it is a prototype version.

The QUIT Command

ProDOS 16 has a single command you can use to leave one application and transfer control to another: QUIT. With it you can either run a specific program or return control to the program whose UserID is on the top of a Quit Return Stack.

The Quit Return Stack is where an application places its UserID if it wishes to regain control the next time an application quits without specifying the pathname of the next application to run. The availability of a Quit Return Stack makes it easy for a supervisory program to execute subsidiary programs so that control always eventually returns to the original program. In fact, the GS program launcher always pushes its UserID on the Quit Return Stack before launching an application. If it did not, you would not return to it when an application ended.

Use the following parameter table with the Quit command:

QUIT ($29)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to next pathname
+4 to +5	flags	I	Return/Restart flags

The pathname parameter points to the pathname of a ProDOS 16 or ProDOS 8 system program. As usual, the pathname begins with a length byte. The current application will load and run this program when you call the QUIT command.

If the pathname pointer is 0, and the Quit Return Stack is not empty, ProDOS 16 pulls a UserID from the Quit Return Stack and executes the program with that ID. If the Quit Return Stack is empty, ProDOS 16 calls up a standard dispatcher subroutine that lets you type in the name of the next system program to run, reboot the disk, or run the SYSTEM/START program.

Only the two high-order bits of the flags parameter are significant. If bit 15 is 1, ProDOS places the current application's UserID on the Quit Return Stack before passing control to the application described by the pathname pointer. This means that control eventually will return to the current application as later programs quit with a 0 pathname parameter. If bit 15 is 0, nothing is placed on the Quit Return Stack.

If bit 14 of the flags is 1, the calling program is capable of being restarted without being reloaded from disk. If control returns to it, it will not be loaded from disk unless it has been purged from memory by the Memory Manager.

The QUIT command never returns control to the application. If an error occurs, an interactive dialog box appears on the screen with the error number on the last line in the box.

INTERRUPT-CONTROL COMMANDS

ProDOS 16 has a 16-entry internal table that contains the addresses of the subroutines it calls when it receives word that a 65816 IRQ interrupt signal has occurred. It calls each subroutine in sequence until one of them claims the interrupt (by clearing the carry flag). If the interrupt is not claimed, a fatal system error occurs.

To use an interrupting device with ProDOS 16, begin by loading its interrupt-handling subroutine into memory. The characteristics of such a subroutine are as follows:

- It must be able to determine if the source of the interrupt is the device for which it is designed.

- If its device is not the source of the interrupt, it must set the carry flag with SEC.

- If its device is the source of the interrupt, it must handle the interrupt by clearing the interrupt condition (usually by reading the device status), performing the necessary I/O operation, and then clearing the carry flag with CLC.

- It must end with an RTL instruction.

The interrupt handler does not have to preserve the status of the A, X, or Y registers since ProDOS restores and saves them.

Next, install the interrupt handler with the ALLOC_INTERRUPT command. The last step is to enable interrupts from the external device.

Here is the structure of the ALLOC_INTERRUPT parameter table:

ALLOC_INTERRUPT ($31)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	int_num	R	Interrupt handler reference number
+2 to +5	int_code	I	Pointer to interrupt handler

In this table, int_num is the reference number returned by ALLOC_INTERRUPT. Int_code is a pointer to the start of the code for the interrupt handler.

Note that if the interrupt handler uses system resources that may be busy (usually ProDOS 16 itself), it should first check the Scheduler's busy flag at $E100FF. If this flag is non-zero, handling of the interrupt must be deferred by adding a task to the Scheduler's queue with the SchAddTask function:

```
PHA                         ;space for result
PushPtr TheHandler          ;Subroutine for Scheduler to call
_SchAddTask
PLA                         ;pop Boolean
```

The Boolean result is true if the task was added to the queue.

HandleInt is the address of a subroutine (ending in RTL) inside the interrupt handler that includes the call to the system resource. When the busy flag is turned off by the interrupted program, the Scheduler automatically calls the HandleInt subroutine to complete processing of the interrupt. (If you don't check the busy flag and ProDOS is busy, you will get an error code of $07 when you attempt to use a ProDOS command.)

To remove an entry from the interrupt handler table, use DEALLOC_INTERRUPT, but only after you have told the external device to stop generating interrupts.

DEALLOC_INTERRUPT uses this parameter table:

DEALLOC_INTERRUPT ($32)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	int_num	I	Interrupt handler reference number

The int_num you pass to DEALLOC_INTERRUPT is the number returned by ALLOC_INTERRUPT when you installed the handler.

STANDARD FILE OPERATIONS

Two file-related operations are so common that Apple Computer, Inc. has developed an entire tool set to support them. The first operation is for displaying a list of disk files so that one can be easily selected. The second one is for requesting the name for a file that is to be saved to disk.

The functions used to perform these operations form part of the Standard File Operations tool set (tool set 23). These functions are called in the same way as any other tool set function. They are not ProDOS 16 commands, so do not call them with the standard ProDOS 16 calling sequence.

The Standard File Operations start-up function is SFStartup. Call it once at the beginning of a program that uses the tool set:

```
PushWord MyID            ;ID of program
PushWord DPAddr          ;Address of one page in bank $00
_SFStartup
```

As you can see, SFStartup requires a direct page in bank $00 to work with.

The shut-down function is SFShutDown. It requires no input parameters and does not return results.

SFGetFile

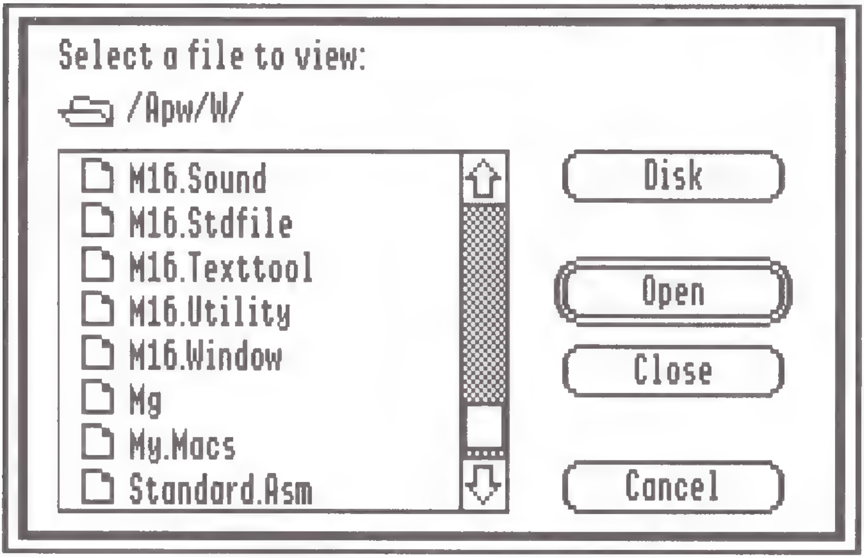
The file-selection function, SFGetFile, was used in the program in listing 10–1. It creates a dialog box similar to the one shown in figure 10–2. At the top of the left half of this box is the name of the current directory; below this name is a window containing an alphabetical list of files in that directory, each preceded by an icon indicating the general file type. The window has a vertical scroll bar that can be used to move any portion of the list of files into view. On the right side are four buttons labeled Open, Close, Cancel, and Disk.

You can select a file by highlighting its name (by clicking on it) and then clicking the Open button, or simply by double-clicking on its name. You can also highlight a name by typing the first character of the name; if there is more than one name beginning with the character, the first one is highlighted. The up- and down-arrow keys can be used to move through the list of files one entry at a time.

If the selected file is a subdirectory (marked by a file-folder icon), the name of the current directory changes and a list of the files in the new, lower-level subdirectory is shown. If the file is not a subdirectory, the dialog box disappears from the screen and the program can deal with the selected file.

It is also possible to display and select files in higher-level subdirectories or on other disks. To move to the next higher subdirectory (closer to the volume directory) click once on the prefix name shown above the file window, type the Esc key, or

Figure 10–2. The SFGetFile Dialog Box



Select a file to view:

/Apw/W/

M16.Sound
M16.Stdfile
M16.Texttool
M16.Utility
M16.Window
Mg
My.Macs
Standard.Asm

click the Close button. Subsequent clicks will eventually take you right up to the volume directory. To display the files on another disk, click the Disk button until a list of the disk's files appears in the file window.

Notice that the first time you click the Disk button, the disk in the current drive is examined to see if you have removed it and inserted another disk; if you have inserted a new disk, the files on it are shown. If you have not, the files on the disk in the next drive are shown. Subsequent clicks will bring you to the other drives in the system.

SFGetFile requires six input parameters, including a pointer to a parameter list called a *reply record*. When SFGetFile ends, the reply record can be read to determine which file, if any, was selected. Here is how to call SFGetFile:

```
PushWord WhereX           ;Horizontal position of box
PushWord WhereY           ;Vertical position of box
PushPtr  Prompt           ;Pointer to prompt string
PushPtr  FilterProc       ;Pointer to filter procedure
PushPtr  TypeList         ;Pointer to file type list
PushPtr  ReplyRec         ;Pointer to reply record
_SFGetFile
RTS
```

```
ReplyRec ANOP                           ;Standard reply record
Good     DS       2                     ;True = open ; False = cancel
Filetype DS       2                     ;File type code
Auxtype  DS       2                     ;Auxiliary type code
Filename DS       16                    ;File name string
Pathname DS       129                   ;Full pathname string

TypeList DC       I1'2'                 ;(byte, not word)
         DC       I1'$04'               ;Text file
         DC       I1'$B0'               ;Source file

Prompt   STR      'Select a file:' ;Prompting string
```

The first two parameters represent the coordinate of the top left-hand corner of the dialog box. The Prompt string is displayed inside the box, just above the file window, and is supposed to be a short message to the user.

Filter Procedure

FilterProc is a subroutine that SFGetFile calls to determine how to display the name of a file or whether to display it at all. FilterProc returns one of three result codes, which SFGetFile examines to make the determination:

0	Do not display the filename
1	Display the filename in dimmed text
2	Display the filename in standard text

Filenames which appear in dimmed text in an SFGetFile dialog box cannot be selected by the user.

If the pointer to FilterProc is 0, SFGetFile does not call a filter procedure, and all filenames are displayed and selectable.

SFGetFile calls the filter procedure in much the same way an application calls a tool set function: it first pushes space for the result code (a word) on the stack, and then pushes a pointer to the file's 39-byte directory entry; finally, it calls the filter procedure with a JSL instruction.

A filter procedure, which you must write yourself, should first decide how the filename is to be displayed. The decision is usually based on the value of the file type code or the filename, both of which are stored in the directory entry record whose address is passed as a parameter. The structure of a directory entry record is shown in table 10–4.

After deciding how to display the filename, the filter procedure must place the appropriate result code in the space reserved for the result on the stack. The address of this space will be the value of the stack pointer plus 8 if the stack pointer has not changed since the procedure was entered.

Table 10–4: The Format of a ProDOS Directory Entry

Offset	*Meaning*
+0 (low four bits)	Length of filename
+0 (high four bits)	Storage type code
+1 to +15	Filename character string
+16	File type code
+17 to +18	Block number of file's index block
+19 to +20	File size (in blocks)
+21 to +23	EOF position
+24 to +25	Creation date
+26 to +27	Creation time
+28	Version of ProDOS which created the file
+29	Lowest version of ProDOS that can use file
+30	Access code
+31 to +32	Auxiliary type code
+33 to +34	Modification date
+35 to +36	Modification time
+37 to +38	Block number of first subdirectory block

Finally, the procedure must remove the input parameter from the stack by moving the three-byte return address on the top of the stack up by four bytes and adding 4 to the stack pointer. When the procedure ends with RTL, the result is on the top of the stack, ready to be popped off by SFGetFile.

Here is an example of a filter procedure you could use to prevent files with a file type code of $FF (ProDOS 8 system programs) from being displayed:

```
          PHD                      ;Save current d.p.
          TSC                      ;Stack pointer to accumulator
          TCD                      ;Align d.p. with stack

          LDY #16                  ;Offset to file type code
          LDA [$6],Y               ;Access the file type code
          AND #$00FF               ;Isolate the byte
          CMP #$FF                 ;ProDOS 8 system program?
          BNE SetCode              ;No, so branch
```

```
         LDA #0                         ;0 = don't include
         BRA SaveCode

SetCode  LDA #2                         ;2 = include/selectable

SaveCode PLD                            ;Restore direct page
         STA 8,S                        ;Save the result

         LDA 2,S                        ;Move the 3-byte return
         STA 6,S                        ; address up by 4 bytes.
         LDA 1,S                        ;(it's now at SP+1 to SP+3)
         STA 5,S

         TSC                            ;Add 4 to the stack pointer
         CLC
         ADC #4
         TCS
         RTL                            ;(Don't use RTS!)
```

This procedure illustrates the standard technique for dealing with pointers that are passed on the stack. After pushing the current value of the direct page register, this routine defines a new direct page that is aligned with the top of the stack. This means the pointer to the directory entry is at position $06 in the direct page (above the two bytes pushed by PHD and the three bytes pushed by the initial JSL to the procedure); because the pointer is now in direct page, you can access the elements of the directory entry with the long indexed indirect addressing mode.

There are two other important matters related to the use of a filter procedure. First, you will not be able to interfere with the display of a subdirectory file; such files are always shown and are selectable. Second, SFGetFile allows a filter procedure to change the contents of the A, X, and Y registers but not the direct page or data bank registers. If you must change the direct page (as in the above example) or the data bank, save it on the stack first and restore it just before the procedure ends.

Type List

TypeList is a table containing a list of file type codes, preceded by a count byte; each file type code occupies one byte in the table. If the filter procedure pointer is 0, SFGetFile displays the name of a file only if the file's type code is in this list. If a filter procedure is defined, filenames referred to in the type list are passed to it for further analysis.

For example, if you want to display only files containing readable text, use the following TypeList table:

```
TypeList  DC I1'2'      ;Two entries
          DC H'04'      ;Textfile (TXT)
          DC H'B0'      ;APW source file (SRC)
```

Directory files are always displayed, even if they are not explicitly included in the TypeList table.

If the pointer to a TypeList table is 0, all types of files will be displayed unless some are restricted by a filter procedure.

You should use a TypeList whenever the list of file type codes you want to permit is relatively short. If you are trying to prevent the display of only a few types of files, it is more convenient to use a filter procedure instead of a Typelist. You will also want to use a filter procedure if you wish to allow restricted file names to be displayed, but not selectable. The filter procedure in listing 10–3 is an example of such a filter.

Reply Record

When the user clicks the Open or Cancel button in the dialog box, control returns to the application and the results are in the reply record. The first field, Good, is a Boolean value indicating which button was pressed; it is true (non-zero) if it was Open or false (zero) if it was Cancel.

The remaining fields have meaning only if Good is true. Filetype and Auxtype hold the file type code and the auxiliary type code of the selected file, respectively. The name of the file is stored at Filename and the full pathname of the file is stored at Pathname. In each case, the name string is preceded by a length byte.

If Open is pressed, SFGetFile sets prefix 0/ (the default directory) to the directory in which the selected file resides. Prefix 0/ does not change if Cancel is pressed.

SFPutFile

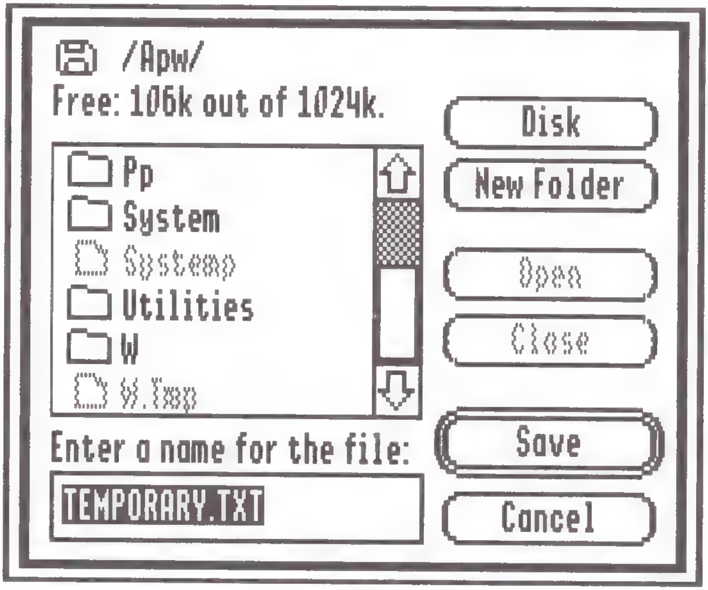
The function to use when you want to request that a filename be entered is SFPutFile (see listing 10–2). This function generates a dialog box similar to the one shown in figure 10–3. Like a SFGetFile box, this dialog box contains a directory window on the left side, but all non-directory files are dimmed. What you are supposed to do is move to the correct subdirectory of the correct disk using the Open, Close, and Disk buttons (much as you would if you were using SFGetFile). Next, you type in the name of the file to be saved in the text-entry rectangle in the lower left corner of the box. This is an EditLine dialog item, so you can use all the standard editing techniques to enter a name. When the SFPutFile dialog first appears, a default filename is highlighted.

It is also possible to create a new subdirectory before selecting a filename. To do this, type in a subdirectory name and then click the New Folder button.

Here is how to call SFPutFile:

```
PushWord WhereX          ;Horizontal position of box
PushWord WhereY          ;Vertical position of box
PushPtr  Prompt          ;Pointer to prompt string
PushPtr  OrigName        ;Pointer to default filename
```

Figure 10–3. The SFPutFile Dialog Box



/Apw/

Free: 106k out of 1024k.

Enter a name for the file:

TEMPORARY.TXT

```
PushWord MaxLen          ;Maximum length of response
PushPtr  ReplyRec        ;Pointer to reply record
_SFPutFile
RTS
```

Many of the same types of parameters used by SFGetFile are also used by SFPutFile. The new parameters used by SFPutFile are OrigName, a pointer to the default filename, and MaxLen, the maximum length of the filename to be entered. In most circumstances, you will want to set MaxLen to 15, the maximum length of a ProDOS filename.

If the name selected (by clicking the Save button) is already being used as a subdirectory name, SFPutFile displays an "I Can't Destroy a Directory" dialog box with an OK button. Another name can be selected after the OK button is clicked. If the name is already being used by a normal data file, SFPutFile displays a "Replace Existing File?" dialog box with Yes and No buttons. If No is clicked, another name can be selected; if Yes is clicked, SFPutFile ends.

SFPutFile also ends when the Cancel button is clicked or when a unique filename is entered and the Save button is clicked. The Good field in the reply record is false if the Cancel button was pressed; otherwise, it is true. If a name was entered, it is returned in the filename field, and the pathname field contains the complete pathname. In addition, the selected directory is assigned to prefix 0/.

REFERENCE SECTION

Table R10–1: The Parameter Tables for the ProDOS 16 Commands

NOTE: In these tables, an Input (I) is a parameter that must be provided before using the command. A Result (R) is a parameter returned by the command. If the Result is a string, you must allocate a space for the string that will be returned and provide a pointer to it.

ALLOC_INTERRUPT ($31)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	int_num	R	Interrupt handler reference number
+2 to +5	int_code	I	Pointer to interrupt handler

CHANGE_PATH ($04)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string
+4 to +7	new_pathname	I	Pointer to the new pathname string

CLEAR_BACKUP_BIT ($0B)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string

CLOSE ($14)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file

CREATE ($01)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string
+4 to +5	access	I	Access code
+6 to +7	file_type	I	File type code
+8 to +11	aux_type	I	Auxiliary type code
+12 to +13	storage_type	I	Storage type code
+14 to +15	create_date	I	Creation date
+16 to +17	create_time	I	Creation time

DEALLOC_INTERRUPT ($32)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	int_num	I	Interrupt handler reference number

DESTROY ($02)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string

FLUSH ($15)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file

FORMAT ($24)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	dev_name	I	Pointer to the device name string
+4 to +7	vol_name	I	Pointer to the volume name string
+8 to +9	file_sys_id	I	Operating system ID code

GET_BOOT_VOL ($28)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	data_buffer	R	Pointer to the volume name string

GET_DEV_NUM ($20)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	dev_name	I	Pointer to the device name string
+4 to +5	dev_num	R	Device reference number

GET_EOF ($19)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	eof	R	The end-of-file position

GET_FILE_INFO ($06)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string
+4 to +5	access	R	Access code
+6 to +7	file_type	R	File type code
+8 to +11	aux_type	R	Auxiliary type code (*)
+12 to +13	storage_type	R	Storage type code
+14 to +15	create_date	R	Creation date
+16 to +17	create_time	R	Creation time
+18 to +19	mod_date	R	Modification date
+20 to +21	mod_time	R	Modification time
+22 to +25	blocks_used	R	Blocks used by the file

(*) For a volume directory file, this field becomes total_blocks (the number of blocks on the volume).

GET_LAST_DEV ($21)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	dev_num	R	Device reference number

GET_LEVEL ($1B)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	level	R	The current file level

GET_MARK ($17)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	position	R	The current mark position

GET_NAME ($27)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	data_buffer	R	Pointer to application name string

GET_PREFIX ($0A)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	prefix_num	I	Prefix number (0 to 7)
+2 to +5	prefix	R	Pointer to prefix name string

GET_VERSION ($2A)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	version	R	ProDOS 16 version number

NEWLINE ($11)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +3	enable_mask	I	Newline enable mask
+4 to +5	newline_char	I	Newline character

OPEN ($10)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	R	Reference number for the file
+2 to +5	pathname	I	Pointer to the pathname string
+6 to +9	null_field	R	Reserved area for ProDOS 16

QUIT ($29)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to next pathname
+4 to +5	flags	I	Return/Restart flags

READ ($12)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for file
+2 to +5	data_buffer	I	Pointer to start of data buffer
+6 to +9	request_count	I	Number of bytes to read
+10 to +13	transfer_count	R	Number of bytes actually read

READ_BLOCK ($22)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	dev_num	I	Device reference number
+2 to +5	data_buffer	I	Pointer to the read buffer
+6 to +9	block_num	I	Block number to read in

SET_EOF ($18)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	eof	I	The new end-of-file position

SET_FILE_INFO ($05)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	pathname	I	Pointer to the pathname string
+4 to +5	access	I	Access code
+6 to +7	file_type	I	File type code
+8 to +11	aux_type	I	Auxiliary type code
+12 to +13	{not used}		
+14 to +15	create_date	I	Creation date
+16 to +17	create_time	I	Creation time
+18 to +19	mod_date	I	Modification date
+20 to +21	mod_time	I	Modification time

SET_LEVEL ($1A)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	level	I	The new file level

SET_MARK ($16)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	position	I	The new mark position

SET_PREFIX ($09)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	prefix_num	I	Prefix number (0 to 7)
+2 to +5	prefix	I	Pointer to the new prefix string

VOLUME ($08)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +3	dev_name	I	Pointer to the device name string
+4 to +7	vol_name	R	Pointer to the volume name string
+8 to +11	total_blocks	R	Size of the volume in blocks
+12 to +15	free_blocks	R	Number of unused blocks
+16 to +17	file_sys_id	R	Operating system ID code

WRITE ($13)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	ref_num	I	Reference number for the file
+2 to +5	data_buffer	I	Pointer to start of data buffer
+6 to +9	request_count	I	Number of bytes to write
+10 to +13	transfer_count	R	Number of bytes actually written

WRITE_BLOCK ($23)

Offset	*Symbolic Name*	*Input or Result*	*Description*
+0 to +1	dev_num	I	Device reference number
+2 to +5	data_buffer	I	Pointer to write buffer
+6 to +9	block_num	I	Block number to be written to

Table R10–2: ProDOS 16 Error Codes

Error Number	*Description*
$00	No error occurred.
$01	The ProDOS command number is invalid.
$07	ProDOS is busy. This error might occur if you try to use ProDOS commands from inside an interrupt handler without first checking that the Scheduler's busy flag at $E100FF is 0.
$10	The specified device cannot be found. ProDOS reports this error after a GET_DEV_NUM command if it cannot locate the device.
$11	The device reference number is invalid. ProDOS reports this error if the device number is not in its list of active devices.

Error Number	*Description*
$25	The ProDOS internal interrupt vector table is full. ProDOS reports this error after an ALLOC_INTERRUPT command if 16 interrupt vectors have already been allocated.
$27	A disk I/O error occurred that prevented the proper transfer of data. If you get this error, the disk is probably irreparably damaged. You will also get this error if there is no disk in a 5 ¼-inch disk drive, however.
$28	The specified disk device is not present. This error occurs if you try to access a second 5 ¼-inch drive when only one drive is present, for example.
$2B	A write operation failed because the disk is write-protected.
$2E	An operation failed because a disk containing an open file was removed from the drive. ProDOS can return this error only for commands that involve checking the volume name.
$2F	The specified device is not connected to the system. This error occurs if there is no disk in a 3 ½-inch drive.
$40	The pathname syntax is invalid because one of the filenames or directory names does not follow the ProDOS naming rules or because a partial pathname was specified and a prefix is not active.
$42	The file control block table is full.
$43	The file reference number is invalid.
$44	The specified directory was not found. This means that one of the subdirectory names in an otherwise valid pathname does not exist.
$45	The specified volume directory was not found. This error occurs when you remove a disk from a drive and then try to access it.
$46	The specified file was not found.
$47	The specified filename already exists.
$48	The disk is full.
$49	The volume directory is full. A volume directory cannot contain more than 51 entries.
$4A	The format of the file specified is unknown or is not compatible with the version of ProDOS being used.

Error Number	*Description*
$4B	The storage type code for the file is invalid.
$4C	An end-of-file condition was encountered during a read operation.
$4D	The mark position is out of range. This error occurs if you attempt to position the mark pointer past the end of the file.
$4E	Access to the file is not allowed. This error occurs when the action prohibited by the access code byte is requested. The access word controls read, write, rename, and destroy operations. The error also occurs if you try to destroy a directory file that is not empty.
$50	The file is already open. You cannot reopen a file that is currently open.
$51	The directory is damaged. ProDOS reports this error when the file count in a directory does not match the number of entries.
$52	The disk is not a ProDOS-formatted (or Apple III SOS-formatted) disk. ProDOS reports this error if it detects a non-ProDOS directory structure.
$53	A parameter is invalid because it is out of range.
$54	Out of memory error. This error is returned by the QUIT command if the ProDOS 8 application specified is too large to fit into the allowable ProDOS 8 memory space.
$55	The volume control block table is full.
$57	Two or more disks with the same volume name are on one line. This is a "warning" error in that ProDOS still performs the operation on the first disk it detects with the volume name required.
$58	The specified device is not a block device. ProDOS supports block-structured devices only.
$59	The level parameter (passed to SET_LEVEL command) is out of range.
$5A	The volume bit map is damaged. ProDOS reports this error if the volume bit map indicates that the disk contains blocks beyond the maximum block count.

Error Number	Description
$5B	Illegal pathname change. This error occurs if the pathnames specified in the CHANGE_PATH command refer to two different volumes. You can only move files between directories on the same volume.
$5C	The specified file is not an executable system file. ProDOS reports this error if you attempt to use QUIT to pass control to a file that is not a ProDOS 16 system file (S16, code $B3) (EXE, code $B5) or a ProDOS 8 system file (SYS, code $FF).
$5D	The operating system specified is not available or not supported. ProDOS returns this error if you use FORMAT to format a disk with an operating system not yet supported by ProDOS 16 or if you try to run a ProDOS 8 system program when the SYSTEM/P8 file is not on the system disk.
$5E	/RAM cannot be removed. This error when you QUIT a ProDOS 8 application if the /RAM RAM disk that uses auxiliary memory (bank $01) cannot be removed.
$5F	Quit Return Stack overflow. ProDOS returns this error if you try to push another program ID on the Quit Return Stack (using the QUIT commamd) when the stack already has 16 entries on it.
$60	It is impossible to determine the last device which was accessed. This error can only be returned by the GET_LAST_DEV command.

NOTE: If a QUIT command results in an error, the error code is not returned to the application. Instead, the code appears in an interactive dialog box on the screen.

Table R10–3: The Major Functions in the Standard File Operations Tool Set ($17)

Function Name	Function Number	Stack Parameters	Description of Parameter
SFGetFile	$09	WhereX (W)	Top left-hand X coordinate
		WhereY (W)	Top left-hand Y coordinate
		PromptStr (L)	Ptr to prompt string
		FilterProc (L)	Ptr to filter procedure

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
		TypeList (L)	Ptr to type list table
		ReplyRecord (L)	Ptr to reply record
SFPutFile	$0A	WhereX (W)	Top left-hand X coordinate
		WhereY (W)	Top left-hand Y coordinate
		PromptStr (L)	Ptr to prompt string
		OrigNameStr (L)	Ptr to default filename
		MaxLen (W)	Maximum length of response
		ReplyRecord (L)	Ptr to reply record
SFShutdown	$03	[no parameters]	
SFStartup	$02	UserID (W)	ID tag for memory allocation
		DPAddr (W)	Address of 1 page in bank 0

Table R10–4: The Major Functions in the Scheduler Tool Set ($07)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
SchStartup	$02	[no parameters]	
SchShutDown	$03	[no parameters]	
SchAddTask	$09	result (W)	Boolean: added to queue?
		TaskPtr (L)	Ptr to interrupt handler

Listing 10–1: A Subroutine for Loading a ProDOS File into Memory

```
LoadFile    START
            Using   FileData
            Using   StartData

; Ask the user for the name of the file to open:

            PushWord #120               ;x
            PushWord #40                ;y
            PushPtr  SFPrompt           ;prompt
            PushLong #0                 ;(no filter)
            PushLong #0                 ;(no file type list)
            PushPtr  ReplyRec           ;reply record
            _SFGetFile

            LDA     good                ;Get the result
            BNE     LoadIt              ;Branch if it was "Open"

            SEC                         ;Set carry if "Cancel"
            RTS

; Load the file into memory:

LoadIt      _OPEN OpenParms             ;Open the file
            LDA     refnum              ;Get reference number
            STA     refnum1             ; and store it in
            STA     refnum2             ; other parameter tables.
            STA     refnum3
            STA     refnum4

            _GET_EOF EOFParms           ;Get size of file

            PHA                         ;space for result
            PHA
            PushLong EOF                ;Push size of file
            PushWord MyID               ;Program ID
            PushWord #$8000             ;Locked
            PushLong #0                 ;(means nothing here)
            _NewHandle

; Dereference the block handle so we can get a pointer for
; the READ command. The handle returned by NewHandle is still
; on the stack, so after PHD/TSC/TCD it's located at
; locations $03-$06 in direct page.

            PHD
            TSC
            TCD
            LDA     [$03]             ;Get pointer (low)
            STA     data_buff
            LDY     #2
            LDA     [$03],Y           ;Get pointer (high)
            STA     data_buff+2

            PLD                       ;Restore d.p.

            PLA                       ;Pop NewHandle result
            PLA

            LDA     EOF               ;Number of bytes to
            STA     request           ; read is set by EOF.
            LDA     EOF+2
            STA     request+2

            _NEWLINE NLParms          ;Disable newline feature
            _READ ReadParms           ;Read the entire file
            _CLOSE CloseParms         ;Close the file

            CLC                       ;Carry clear means "Open"
            RTS

            END

FileData    DATA

; SFGetFile data:

SFPrompt    STR     'Select a file to view:'

ReplyRec    ANOP
good        DS      2                 ;Non-zero if open pressed
filetype    DS      2                 ;ProDOS file type
auxtype     DS      2                 ;ProDOS auxiliary file type
FileName    DS      16                ;Name of file in directory 0/
fullpath    DS      129               ;Full pathname

; Data for file I/O operations:

OpenParms   ANOP
refnum      DS      2                 ;Reference number returned here
            DC      I4'FileName'      ;Pointer to filename
            DS      4                 ;Space for I/O buffer pointer

EOFParms    ANOP
refnum1     DS      2
EOF         DS      4                 ;EOF returned here
```

```
NLParms     ANOP
refnum2     DS      2
            DC      I2'0'                 ;disable newline read mode
            DS      2

ReadParms   ANOP
refnum3     DS      2
data_buff   DS      4                     ;Pointer to data area
request     DS      4                     ;Number of bytes to read
            DS      4                     ;(number actually read)

CloseParms  ANOP
refnum4     DS      2

            END

StartData   DATA

MyID        DS      2                     ;Store MMStartup result here

            END
```

Listing 10–2: A Subroutine for Saving a Block of Memory to Disk as a ProDOS File

```
; This subroutine asks the user to enter a filename,
; then saves a block of data to that file on disk.
;
; On entry, data_buff/data_buff+2 points to the start
; of the data buffer. The length of the buffer is
; stored at actual/actual+2. These variables are
; defined in the FileData data segment.

SaveFile    START
            Using   FileData

; Ask the user to enter a name for the file:

            PushWord #120                 ;x
            PushWord #40                  ;y
            PushPtr  SavePrompt           ;prompt
            PushPtr  FN_Default           ;default filename
            PushWord #15                  ;15 characters max
            PushPtr  ReplyRec             ;reply record
            _SFPutFile

            LDA     good                  ;Get the result
            BNE     SaveIt                ;Branch if it was "Save"

            SEC                           ;Set carry if "Cancel"
            RTS
```

```
SaveIt     ANOP

;Set up a pointer to the area to be saved:

           LDA     data_buff
           STA     w_buffer
           LDA     data_buff+2
           STA     w_buffer+2

;Set up the number of bytes to be saved:

           LDA     actual
           STA     w_request
           LDA     actual+2
           STA     w_request+2

           _CREATE CreatePrms         ;Create the file
           _OPEN   OpenParms          ;Open the file
           LDA     refnum
           STA     w_refnum
           STA     c_refnum

           _WRITE  WriteParms         ;Write data
           _CLOSE  CloseParms

           CLC                        ;Clear carry if "Save"
           RTS

           END

FileData   DATA

; SFPutFile data:

ReplyRec   ANOP
good       DS      2                  ;Non-zero if "Save" pressed
           DS      2                  ;[not used]
           DS      2                  ;[not used]
FileName   DS      16                 ;Filename selected
fullpath   DS      129                ;Full pathname in 0/

SavePrompt STR     'Enter a name for the file:'

FN_Default STR     'TEMPORARY.TXT'   ;Default filename

; Data for file I/O operations:

CreatePrms ANOP
           DC      I4'FileName'       ;Pointer to filename
           DC      I2'$E3'            ;access code
           DC      I2'$04'            ;TXT file
```

```
            DC      I4'0'               ;aux type
            DC      I2'1'               ;storage type
            DC      I2'0'               ;creation date
            DC      I2'0'               ;creation time

OpenParms   ANOP
refnum      DS      2                   ;Reference number returned here
            DC      I4'FileName'        ;Pointer to filename
            DS      4                   ;Space for I/O buffer pointer

WriteParms  ANOP
w_refnum    DS      2
w_buffer    DS      4                   ;Pointer to data area
w_request   DS      4                   ;Number of bytes to write
w_actual    DS      4                   ;(number actually written)

CloseParms  ANOP
c_refnum    DS      2

            END
```

Listing 10–3: The Structure of an SFGetFile Filter Procedure

```
**********************************************
* This is a filter procedure for SFGetFile.  *
*                                            *
* It makes only TXT and SRC files selectable *
* from a file list; all other file names are *
* dimmed.                                    *
*                                            *
* On entry, here is what is on the stack:    *
*                                            *
*     result (word)                          *
*     pointer to directory entry (long)      *
*     return address (3 bytes)               *
**********************************************
SF_Filter   START

; Direct page offsets:

OldDP       EQU     $01
RetAddr     EQU     OldDP+2
DirEntry    EQU     RetAddr+3
Result      EQU     DirEntry+4

            PHD                         ;Save direct page
            TSC
            TCD                         ;Align d.p. with stack

            LDY     #16                 ;Offset to file type code
            LDA     [DirEntry],Y
```

```
          AND      #$00FF            ;(use low byte only)
          CMP      #$B0              ;SRC file?
          BEQ      FP0               ;Yes, so branch
          CMP      #$04              ;TXT file?
          BEQ      FP0               ;Yes, so branch

          LDA      #1                ;1 = display/not selectable
          BRA      FP1

FP0       LDA      #2                ;2 = display/selectable

FP1       STA      Result            ;Save the result

          PLD                        ;Restore d.p.

          LDA      2,S               ;Move 3-byte return
          STA      6,S               ; address up by 4 bytes.
          LDA      1,S
          STA      5,S

          TSC                        ;Add 4 to the
          CLC                        ; stack pointer.
          ADC      #4
          TCS
          RTL

          END
```

CHAPTER 11

Sound and Music

One of the remarkable features of the Apple IIGS is its ability to create incredibly realistic sound effects and music. The GS can do this because its sound system is controlled by the powerful Ensoniq Digital Oscillator Chip (DOC), the same chip used in the professional-quality Mirage Digital Sampling Keyboard and the Ensoniq Piano.

With appropriate software, the Ensoniq DOC can generate up to 32 user-definable sounds simultaneously. The frequency of these sounds can be varied over a wide range and the volume can be set to any of 256 discrete levels. As a result, it is possible to mimic closely the acoustical behavior of any musical instrument. The DOC is also capable of sampling incoming audio signals, including voice samples, and converting them to digital form.

Note: To maintain compatibility with earlier Apple II models, the GS also lets you create simple sounds by toggling the GS speaker with a soft switch at $E0C030. You can use the SysBeep function in the Miscellaneous Tool Set to beep the speaker using this technique. SysBeep requires no parameters.

This chapter takes a close look at the Ensoniq DOC and examines ways of manipulating its internal registers to create sound and music. It also discusses using the Sound Manager to communicate with the DOC and to play back prerecorded sound patterns. Finally, the Note Synthesizer's ability to play musical tunes using different simulated instruments is covered.

SOUND HARDWARE

A block diagram of the GS sound system is shown in figure 11–1. All sound-control commands are sent by a program running in the 65816 address space to the Sound General Logic Unit (GLU). The GLU passes them on to the sound circuitry controlled by the Ensoniq DOC. Because the Ensoniq is isolated from the 65816 by

Figure 11–1. The Apple IIGS Sound System Block Diagram

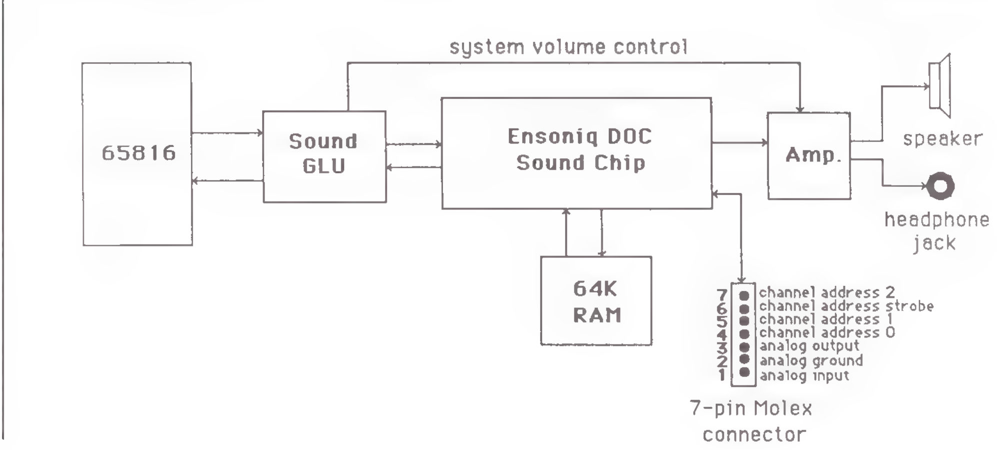


Table 11–1: The Sound GLU Registers

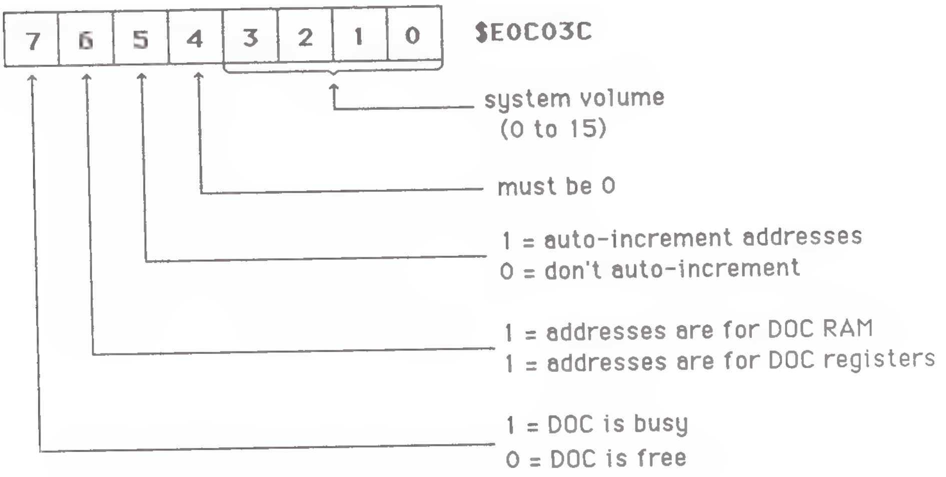
Register Name	*Address*
Sound Control	$C03C
Data	$C03D
Address Low	$C03E
Address High	$C03F

the GLU in this way, it can generate sound independently of the current application running in memory. That is, the application can perform other tasks while sound is playing in the background.

Sound GLU

The Sound GLU contains four registers for communicating with the DOC (see table 11–1). The Sound Control register shown in figure 11–2 is the most complex. The lower four bits control the output volume level of the audio amplifier. The GS sound tools put the volume level (as set by the Control Panel) into these bits before they access the DOC.

Figure 11–2. The Sound Control Register



Bit 7 is a status bit that indicates whether the DOC is ready to respond to read and write operations. It is ready only if this bit is 0.

Bit 6 controls whether the value in the Sound GLU 16-bit address register refers to an address in the DOC's special 64K RAM waveform buffer (bit = 1) or to a DOC register number (bit = 0).

Bit 5 controls whether the GLU address register is to increment automatically after a data register access (bit = 1) or whether it is to stay the same (bit = 0). Setting the auto-increment bit makes it easier to access a range of registers or RAM addresses.

The 16-bit address register contains either an address in the DOC RAM area or a DOC register number, depending on the state of bit 6 of the Sound Control register. For register accesses, the high-order part of the address register is ignored.

The data register can be used to write data to DOC registers or to DOC RAM, as well as for reading data. Here is how to read or write a byte of data:

1. Adjust the value in the Sound Register to indicate whether a DOC register or DOC RAM is to be accessed, and set the auto-increment bit to the desired state. The volume bits should be set to the system-volume value (it is stored in the low-order four bits of $E100CA).

2. Store the DOC register number, or the DOC RAM address, as the case may be, in the address registers.

3. Store the data byte in the data register (write operation) or load the data byte from the data register (read operation).

One caveat should be mentioned that affects read operations involving the DOC registers or DOC RAM. A read operation immediately following the storing of an address in the Sound GLU address register lags by one cycle, so you must always ignore the first result returned. If auto-increment is on, you need only ignore the first result in a sequence of successive read operations.

A final note on the Sound GLU DOC registers: Apple has warned developers not to access these registers directly, because their functions or addresses may change in future versions of the GS. If you need to manipulate the DOC registers or DOC RAM, you should use the Sound Manager functions described later in this chapter.

Ensoniq DOC

The Ensoniq DOC is the heart of the GS sound system. It contains 32 independent oscillators (numbered from 0 to 31) that can be used to sample, at user-definable rates, digitized audio waveforms stored in memory. This sampling rate indirectly sets the frequency of the sound generated by the oscillator. The DOC has a total of 227 internal registers for controlling the operation of its oscillators and of the DOC chip itself.

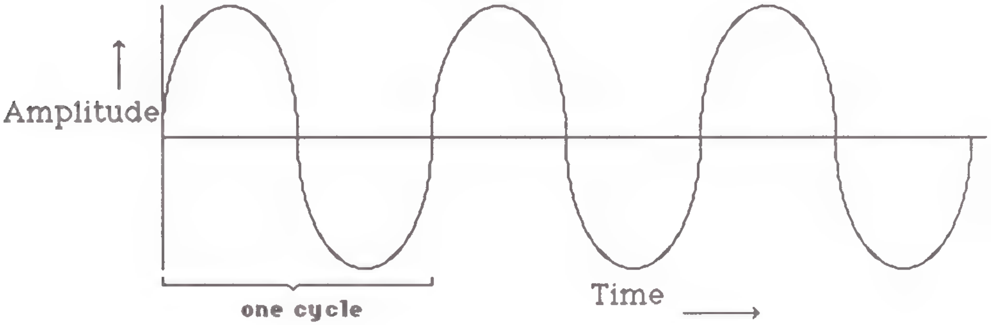
Because each oscillator operates independently of any other, you can play 32 waveforms simultaneously! The GS sound tools pair these oscillators off to create 16 *voices* or *generators*. This is done to make it possible to create richer and more realistic sounds. One of these generators (corresponding to oscillators 30 and 31) is reserved for use as a general-purpose timer by the GS sound tools.

The output signal of the DOC goes to an analog circuit that amplifies the signal before sending it to the GS speaker (or other audio output device). The gain of the amplifier is the system volume set by the Sound Control register in the Sound GLU.

The audio waveforms are stored in a 64K RAM memory area dedicated to the DOC. This RAM does not form part of the 65816 address space and can be accessed only through the Sound GLU registers.

An audio signal is called a *waveform*, because if you plot a graph of signal intensity (amplitude) versus time for a pure tone, you get a series of sinusoidal waves, as shown in figure 11–3. The frequency (or *pitch*) of the signal is expressed in cycles per second, or *Hertz*. As far as the DOC is concerned, however, the waveform need not be a symmetric wave at all. It could have a completely random form (white noise) or a complex, but periodic, form made up of a combination of sinusoidal waves. By playing with the waveform you can generate all sorts of interesting sound

Figure 11–3. Audio Waveforms



variations; in fact, one of the reasons different musical instruments do not sound exactly the same is that the waveforms they generate for the same pitch are different.

A digitized waveform stored in the DOC RAM area is a series of bytes representing the amplitude of the sound wave at fixed time intervals. The bytes can take on values from $01 (low volume) to $FF (high volume). A DOC oscillator stops when it encounters a $00 byte.

Sound Output

The DOC steps through a waveform at a user-definable rate and feeds the bytes it reads to a built-in digital-to-analog converter. The analog output from the converter goes to four different destinations:

- The built-in two-inch speaker
- The RCA mini headphone/speaker output jack. You can connect Walkman-style headphones or (with an adapter) the input line of your stereo set to this jack
- Pin 3 of the 7-pin Molex connector used by the DOC
- Pin 11 of the RGB video connector (this is for the benefit of RGB monitors with internal speakers)

Note that if the headphone/speaker jack is in use, output to the speaker is automatically disabled.

Software that creates sound using the DOC can assign the sound to any of eight output channels. (The DOC actually can handle up to sixteen channels, but the GS supports only eight of them.) The channel address appears on pins 4, 5, and 7 of

Table 11–2: The 227 Ensoniq DOC Registers

Register Name	*Register Number*
Frequency Low	$00 + OSC
Frequency High	$20 + OSC
Volume	$40 + OSC
Data	$60 + OSC
Address	$80 + OSC
Control	$A0 + OSC
Waveform	$C0 + OSC
Oscillator Interrupt	$E0
Oscillator Enable	$E1
Analog-to-Digital Converter	$E2

OSC = the oscillator number ($00 to $1F). These are 8-bit registers.

the Molex connector. An external demultiplexer can monitor this address and direct the output signal (on pin 3) to a different speaker for each channel. The MDIdeas SuperSonic stereo card, for example, works with two speakers, sending all sound for even-numbered channels to one of them and all sound for odd-numbered channels to the other.

THE ENSONIQ DOC REGISTERS

Each of the 32 oscillators in the Ensoniq DOC has seven 8-bit registers associated with it (see table 11–2). Two of these registers set the waveform sampling rate (frequency) of the oscillator; the others set the volume, the position of the oscillator's waveform in the DOC RAM area, the characteristics of the waveform, and the oscillator's operational mode. Finally, one register contains the last byte the oscillator loaded from the waveform table.

There are also three general-purpose DOC registers you can use to enable oscillators, read interrupt status information, and perform analog-to-digital conversion of incoming audio signals.

In total, there are 227 registers inside the Ensoniq DOC chip. The sections that follow examine precisely how to use them.

Oscillator Registers

This section discusses the seven registers assigned to each oscillator. Keep in mind that, as shown in table 11–2, the register number for a particular oscillator is the sum of a base register number and the oscillator number. For example, the base register number for Volume is \$40; therefore, the Volume register for oscillator \$1C is \$5C (\$40 + \$1C).

Frequency Low (\$00-\$1F) and Frequency High (\$20-\$3F). The DOC's internal 16-bit frequency register is a concatenation of the Frequency Low and Frequency High registers. The value stored in this combined register sets the speed at which the oscillator retrieves data from the waveform in memory. This speed, called the ***sampling rate***, is given by the following formula:

$$\text{Sampling rate} = \frac{(\text{Scan Rate}) * \text{F}}{2^{(17+\text{RES})}} \text{ samples/second}$$

where

$$\text{Scan Rate} = \frac{7.159 * 10^6}{(8 * (\text{OSC}+2))} \text{ /second}$$

In these formulas, F represents the value stored in the 16-bit frequency register and OSC represents the number of enabled oscillators. RES is the number stored in the resolution bits of the Waveform register (see below).

Note that the sampling rate is not the same as the frequency of the output signal to the speaker. This frequency depends on the size of the waveform being played by the oscillator. To determine the frequency of the speaker signal, divide the sampling rate by the number of bytes needed to define one cycle of the waveform.

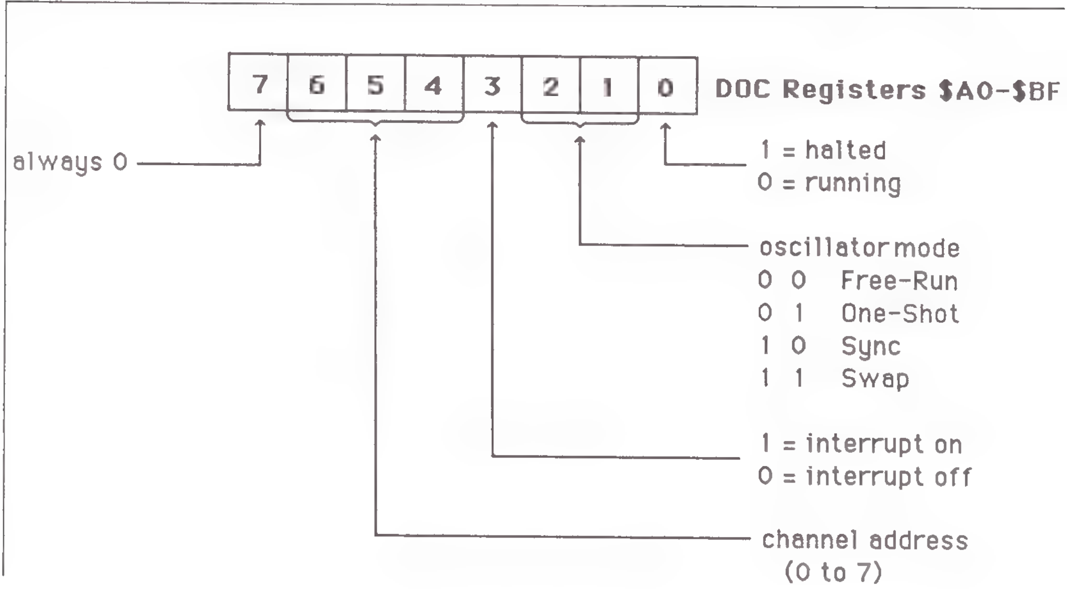
Volume (\$40-\$5F). The Volume register sets the volume of the oscillator. When the DOC reads a waveform data byte from memory, it multiplies it by the value in the Volume register before passing it to the output amplifier.

Data (\$60-\$7F). The Data register contains the last byte the DOC read from the waveform for the oscillator.

Address (\$80-\$9F). The Address register contains the page number inside the DOC RAM at which the waveform for the oscillator begins. The waveform must start on a page boundary.

Control (\$A0-\$BF). The Control register is used to set the operational mode of the oscillator. It controls the output channel with which the oscillator works, whether

Figure 11–4. The Oscillator Control Register



interrupts are enabled, the oscillator mode, and whether the oscillator has been halted or is playing.

The meanings of each of the bits in the Control register is shown in figure 11–4. The channel address is three bits wide so there are eight possible output channels. As shown in figure 11–1, the channel address lines are brought out at the 7-pin Molex connector. By connecting a multiplexer to this connector, you can easily generate multiphonic sound.

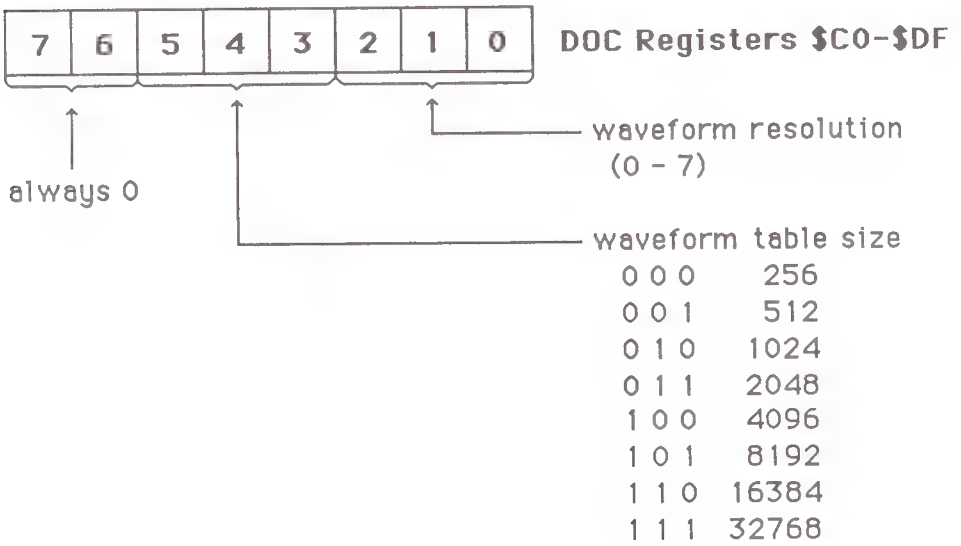
If interrupts are enabled for the oscillator, the interrupt flag in the Oscillator Interrupt register is set when the oscillator finishes playing its waveform. In addition, the oscillator number is stored in bits 1–5 of the Oscillator Interrupt register.

The halt bit controls whether the oscillator is playing. When it is 1, the oscillator is halted and no sound is generated. When it is 0, the oscillator begins playing its waveform.

Four different oscillator modes can be used:

> Free-run mode (0). In free-run mode, the oscillator plays the waveform again and again until the halt bit of the Oscillator Control register is set by the application. (The oscillator also halts if it encounters a $00 byte in the waveform table.)

Figure 11–5. The Waveform Register



One-shot mode (1). In one-shot mode, the oscillator plays the waveform once and then sets its halt bit and stops. This is the mode you would use if the waveform represents a long, non-periodic sound.

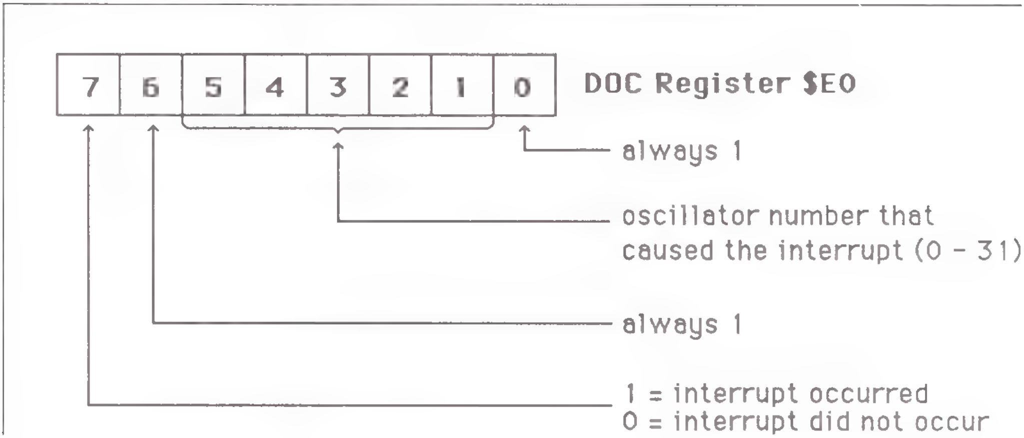
Sync mode (2). In sync mode, a pair of adjacent even/odd oscillators work together in synchronization. (The odd-numbered oscillator always has the higher number of the two.) In particular, when the even-numbered oscillator begins playing its waveform, the odd-numbered one begins playing its waveform as well.

Swap mode (3). In swap mode, as in sync mode, a pair of adjacent even/odd oscillators work together, but in a different way. First the enabled oscillator runs in one-shot mode. When it finishes stepping through its waveform table, it sets its halt bit and then clears the halt bit of the other oscillator so that it will begin playing its waveform. An even/odd pair of oscillators in swap mode is called a *generator*.

Waveform ($C0-$DF). The Waveform register contains the size of the waveform being played by the oscillator. It also tells the DOC the resolution of the waveform.

As shown in figure 11–5, the size of the waveform is called the *Table Size* and is represented by a 3-bit code number. These codes correspond to sizes of 256, 512,

Figure 11–6. The Oscillator Interrupt Register


7
6
5
4
3
2
1
0
DOC Register $E0
always 1
oscillator number that caused the interrupt (0 - 31)
always 1
1 = interrupt occurred
0 = interrupt did not occur

1024, 2048, 4096, 8192, 16384, and 32768 bytes, respectively. No other sizes are permitted.

The waveform resolution is also represented by a 3-bit code number from 0 to 7. Earlier in this section the ways in which this value participates in the waveform sampling rate calculation were discussed. If the resolution is raised by 1, the sampling frequency is halved; similarly, if it is lowered by one, the sampling frequency doubles. In most cases, the resolution code should be set equal to the Table Size code so that the effective frequency of the signal sent to the speaker is not affected by powers of two.

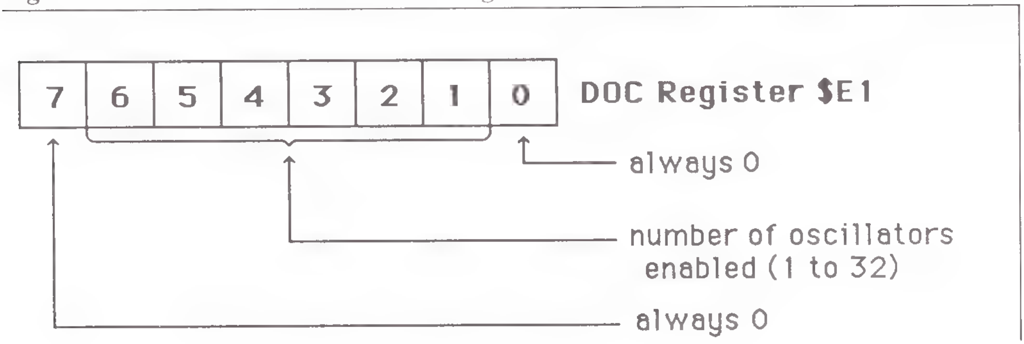
General Registers

The following three registers in the DOC are of general use and do not relate to any specific oscillator.

Oscillator Interrupt ($E0). The Oscillator Interrupt register indicates whether an interrupt has occurred and, if so, which oscillator caused it (see figure 11–6). An interrupt will occur when an oscillator finishes playing its waveform if the interrupt enable bit in the oscillator's Control register is set to 1. Bit 7 contains the interrupt flag, and bits 1–5 contain the number of the oscillator that caused the interrupt (0 to 31).

Oscillator Enable ($E1). This register controls the number of enabled oscillators (see figure 11–7). This number can range from 1 to 32 and is stored in bits 1–5 of the register. (The minimum is not 0 because at least one oscillator must be enabled.) Oscillators are enabled in numeric order, starting with #0.

Figure 11–7. The Oscillator Enable Register



Note that the number of enabled oscillators affects the rate at which the DOC scans the oscillator wavetables. The more oscillators are enabled, the slower the rate. To simplify calculations, the GS sound tools enable all 32 oscillators.

Analog-to-Digital Converter ($E2). This register contains the output of an analog-to-digital converter whose input line is connected to pin 1 of the Molex connector used by the Ensoniq chip (see figure 11–1).

To read the result of the analog-to-digital conversion and to initiate the next conversion, all you have to do is read this register. The conversion process takes about 31 microseconds, so do not read the register again until this time has elapsed. If you do, the result read will be meaningless.

SOUND MANAGER

The Sound Manager (tool set 8) lets you manipulate the DOC registers and DOC RAM in such a way that you do not have to concern yourself with the mechanics of the Sound GLU or the I/O addresses it uses. It also defines and lets you use a free-form synthesizer which is useful for playing back digitized sound effects or one long musical piece.

The Sound Manager deals with oscillators in swap mode pairs called *generators*. Sixteen generators are defined, but one is reserved for use as a timer.

Before using the Sound Manager, start it up by calling SoundStartup:

```
PushWord DPAddr          ;Pointer to one-page in bank $00
_SoundStartup
```

The direct page work area whose address is passed to SoundStartup must be page-aligned.

When you are through with the Sound Manager, call SoundShutDown (no parameters). It shuts off all the DOC oscillators.

Accessing the DOC RAM Area

Before you can create sound with the DOC, there must be a waveform in the DOC RAM. You can put one there with WriteRamBlock:

```
             PushPtr  MyWave              ;Pointer to wave data
             PushWord #$0000              ;Starting address in DOC RAM
             PushWord #$100               ;Size of wave in bytes
             _WriteRamBlock
             RTS

    MyWave   DS       256                 ;Insert wave definition here.
```

The low-order byte of the starting address passed to WriteRamBlock should be $00, because the DOC expects waves to begin on page boundaries. The wave size should be $100, $200, $400, $800, $1000, $2000, $4000, or $8000, because these are the only sizes the DOC supports.

The waveform can contain any series of one-byte values, but it must not contain a $00 byte, because a DOC oscillator stops playing when it encounters a 0 value. The mid-range value of $80 corresponds to a 0 output voltage. Listing 11–1 shows how to create two types of simple one-page waveforms, a sinusoidal wave and a triangular-shaped wave.

To read data from DOC RAM to main memory, use ReadRamBlock:

```
    PushPtr MyBuffer          ;Pointer to buffer in main memory
    PushWord #$1000           ;Starting address in DOC RAM
    PushWord #$400            ;Number of bytes to transfer
    _ReadRamBlock
```

Volume Control

The Sound Manager has two functions related to volume settings: GetSoundVolume and SetSoundVolume. They can be used to read or set the volume level of any generator or the system volume.

To read the volume, call GetSoundVolume as follows:

```
    PHA                         ;space for result
    PushWord GenNum             ;generator number
    _GetSoundVolume
    PLA                         ;Pop the volume setting
```

If GenNum is between 0 and 14, it represents a generator number, and the volume number for that generator is returned on the stack. The volume number can range from 0 to 255.

If GenNum is greater than 14, the system volume is returned. The system volume ranges from 0 to 15 only and is returned in bits 4–7 of the result.

You can set the volume of a particular generator, or the system volume, with SetSoundVolume:

```
PushWord VolLevel          ;Volume level (0 to 15)
PushWord GenNum            ;Generator number
_SetSoundVolume
```

As with GetSoundVolume, if the generator number is above 14, the system volume is affected and the volume code must be stored in bits 4–7 of VolLevel.

Free-form Synthesizer

The Sound Manager supports a free-form synthesizer you can use to play back any digitized sound stored in the DOC RAM area. It is intended for the playback of complex, non-repeatable waveforms that have previously been digitized. To play musical notes, it is much more convenient to use the Note Synthesizer, as we will see below.

The main free-form synthesizer function is FFStartSound. It allocates a particular generator and causes it to start playing a waveform stored in memory. Here is how to call it:

```
         PushWord #GenNum*256+$01   ;Generator (high), mode (low)
         PushPtr FF_Parms           ;pointer to parameter block
         _FFStartSound
         RTS

FF_Parms ANOP
         DC          I4'WaveForm'   ;Pointer to waveform
         DC          I2'WaveSize'   ;Size of waveform in pages
         DC          I2'Frequency'  ;Playback rate
         DC          I2'DOC_Addr'   ;DOC position for wave
         DC          I2'BufferSize' ;DOC wave size code
         DC          I4'NextWave'   ;Pointer to next parm table
         DC          I2'Volume'     ;Generator volume (0-255)
```

The first parameter passed to FFStartSound contains the generator number (0 to 14) in the high-order byte and a mode code of $01 in the low-order byte. The value $01 means the generator is used for free-form synthesizer mode.

The parameter table tells the free-form synthesizer all it needs to know about the waveform to be played. WaveForm is the address of the waveform in the 65816 memory space and WaveSize is its size in 256-byte pages.

The Frequency field contains the number to be placed in the DOC's frequency register for the two oscillators in the generator; this value can be determined using the following formula:

$$\text{Frequency Register} = \frac{32 * \text{Sampling Rate}}{1645}$$

The sampling rate is expressed in cycles/second (Hertz) in this formula.

DOC_Addr is the starting address in the DOC RAM area to which the waveform is to be transferred before it is played. BufferSize is a code from 0 to 7 representing the Table Size of the form. (See the discussion of Table Size in connection with the DOC Waveform registers earlier in this chapter.)

NextWave is the address of the parameter block for the waveform to be played when the current waveform has been finished. Set it to 0 for the last parameter block. If you want to play one wave over and over again, set the NextWave parameter to the address of the one and only parameter table. (You can stop the sound with FFStopSound.)

Volume is the volume of the waveform and can range from 0 (low) to 255 (high).

Four other major functions can be used to control the free-form synthesizer or to check its status:

- FFStopSound stops one or more generators
- FFSoundStatus checks the status of all generators
- FFGeneratorStatus checks the status of one generator
- FFSoundDoneStatus determines if the synthesizer is playing a waveform

Two minor functions, SetSoundMIRQV and SetUserSoundIRQV, are used to install special sound interrupt handlers and are rarely used by applications. They are not discussed here.

You can use FFStopSound to stop any generators that may still be running:

```
PushWord GenMask            ;Generator mask
_FFStopSound
```

Each generator corresponds to a unique bit in the generator mask: bit 0 for generator #0, bit 1 for generator #1, and so on. To stop a particular generator, set its bit in the mask to 1. For example, to stop all 15 usable generators, use a mask of $7FFF. Bit 15 of the mask corresponds to the reserved generator, so it must always be 0 in the mask.

To determine which generators are playing at any given time and which are idle, use FFSoundStatus:

```
PHA                         ;Space for result
_FFSoundStatus
PLA                         ;Pop the generator mask
```

FFSoundStatus returns a generator mask that is of the same form as the one passed to FFStopSound. If a bit is set to 1, the generator corresponding to that bit is currently playing.

FFGeneratorStatus returns the first two bytes of the generator control block (GCB) for the generator. These bytes contain the synthesizer mode, generator number, and channel number.

The GCB is a 16-byte data structure used by the free-form synthesizer to keep track of the attributes and status of the generator. The GCB's for each of the sixteen generators are stored in the one-page direct page area whose address is passed to SoundStartup.

To call FFGeneratorStatus, pass it the number of the generator in question:

```
PHA                         ;space for result
PushWord GenNumber          ;Generator number
_FFGeneratorStatus
PLA                         ;Pop the status word
```

Bits 0–3 of the status word contain the generator number, bits 4–7 contain the output channel number (normally 0) and bits 8–11 contain the mode code. The mode code is $1 for the free-form synthesizer. Bit 15 is set to 1 if the last block of the waveform has been loaded.

The last major free-form function is FFSoundDoneStatus. Call it to determine if a free-form synthesizer generator is currently playing anything:

```
PHA                         ;space for result
PushWord GenNumber          ;generator number
_FFSoundDoneStatus
PLA                         ;Busy code
```

If the result is $0000, the generator is still playing; if it is $FFFF, it is done.

Low-level Access to DOC Registers and DOC RAM

As mentioned earlier in this chapter, you should not use the Sound GLU registers to communicate directly with the Ensoniq DOC registers or RAM. Instead, you should use the Sound Manager's GetTableAddress function to get the address of a table containing the addresses of six low-level DOC control subroutines; you should then call these subroutines to communicate with the DOC. GetTableAddress also returns the addresses of two useful oscillator/generator translation tables and a table of GCB offset addresses.

Here is how to call GetTableAddress:

```
PHA                         ;space for result
PHA
_GetTableAddress
PopLong DOC_Table           ;Pop pointer to table
```

Table 11-3: The Low-level DOC Subroutine Table Whose Address Is Returned by GetTableAddress

Subroutine or Table Pointer	*Offset*
Read Register	$00
Write Register	$04
Read RAM	$08
Write RAM	$0C
Read Next	$10
Write Next	$14
Oscillator Table	$18
Generator Table	$1C
GCB Table	$20

DOC_Table should be a direct page variable, so that you can use it to indirectly access the contents of the table.

The format of the table whose address is returned by GetTableAddress is shown in table 11–3. It contains nine 4-byte addresses (low-order bytes first) of low-level DOC subroutines or data tables.

The first six entries in the table are 4-byte addresses of subroutines (low-order bytes first) that can be called with a JSL instruction. Unfortunately, the 65816's indirect long addressing mode is useful only if you know in advance where the address table is located, and you do not have this information. The easiest way to call a subroutine is to store its address in the 3-byte operand of a JSL instruction, low-order byte first. This involves ugly self-modifying code, but there is no easy alternative.

For example, here is how to set up a JSL to the write subroutine (at offset $04):

```
TablePtr    GEQU $00                    ;pointer to low-level table

            PHA                         ;space for result
            PHA
            _GetTableAddress            ;Get table pointer
            PopLong TablePtr
```

```
           LDA [TablePtr]          ;Get address (low)
           CLC
           ADC #$04                ;... add offset
           STA JSL_Patch+1         ;Set low word of operand
           LDY #2
           LDA [TablePtr],Y        ;Get address (high)
           ADC #0                  ;(adjust for any carry)

           SEP   #$20              ;8-bit accumulator
           LONGA OFF

           STA JSL_Patch+3         ;Save bank byte

           REP   #$20              ;16-bit accumulator
           LONGA ON

           RTS

JSL_Patch  JSL $123456             ;Call low-level subroutine
           RTS
```

Before a DOC subroutine is called, the 65816 must be in native mode with 16-bit index registers (x=0) and an 8-bit accumulator (m=1). If you are in full native mode, you can switch to an 8-bit accumulator with the following assembler instructions:

```
SEP   #$20
LONGA OFF
```

After you call the subroutine, you can switch back to a 16-bit accumulator with a REP #$20 instruction, followed by a LONGA ON directive.

The Oscillator Table entry is a pointer to a 16-byte generator-to-oscillator translation table. This table contains a list of one-byte oscillator numbers, in generator-number order. The second oscillator for a given generator is always the next higher-numbered one. For example, to determine the two oscillators for generator #9, you would use a code fragment like this:

```
OscTbl  GEQU    $04

        LDY     #$18
        LDA     [TablePtr],Y    ;Get pointer (low)
        STA     OscTbl
        LDY     #$1A
        LDA     [TablePtr],Y    ;Get pointer (high)
        STA     OscTbl+2

        SEP     #$20            ;Switch to 8-bit accumulator
        LONGA   OFF
        LDY     #9                  ;Generator #9
        LDA     [OscTbl],Y          ;Get first oscillator
        STA     Osc1                ;Save number
        INC     A                   ;Second oscillator = first + 1
        STA     Osc2                ;Save number

        REP     #$20                ;Back to 16-bit accumulator
        LONGA   ON
        RTS
```

The Generator Table entry points to a 32-byte table you can use to determine which oscillators are associated with which generators. It contains a list of generators in oscillator-number order.

The GCB Table contains the one-byte offset positions to the generator control blocks for the 15 usable generators, in generator-number order. The base point for the offsets is the address of the bank $00 page passed to SoundStartup. Most applications have no need to access the GCB.

The methods that can be used to call the six low-level DOC subroutines are discussed in the sections below.

Read Register. Call this subroutine to read the contents of any of the 227 registers in the DOC. The register number must be in the X register. The contents of the specified DOC register is returned in the 8-bit accumulator.

Write Register. This subroutine stores a number in any DOC register. The register number must be in the X register, and the 8-bit accumulator contains the number to be written.

Read RAM. This subroutine reads the value stored at any location in the DOC RAM area. On entry, the X register must contain the address. The value is returned in the 8-bit accumulator.

Write RAM. This subroutine writes the value in the 8-bit accumulator to any location in the DOC RAM area. On entry, the address to be written to must be in the X register.

Read Next. If the previous low-level call was Read Register or Write Register, this subroutine returns in the 8-bit accumulator the data in the next DOC register. If the previous call was Read RAM or Write RAM, it returns the data in the next RAM location.

Write Next. This subroutine is similar to Read Next, but it returns in the 8-bit accumulator the contents of the next register or the next RAM location, as the case may be.

Using Low-level Subroutines: Digitizing Analog Input

The CONVERTER program in listing 11–2 illustrates how to call low-level DOC subroutines. It digitizes an incoming analog signal (which appears on pin 1 of the Ensoniq's Molex connector) by reading DOC register $E2 again and again until a 32K buffer fills up or until ESC is pressed. (To play the sound back, you would transfer the sample to DOC RAM and call FFStartSound.)

You can use a device such as the MDIdeas SuperSonic stereo card (with Digitizer option) as the source of the analog signal to be digitized. The SuperSonic receives its signal through an RCA phono connector, which you can connect to the output line of a radio, tape recorder, record player, CD player, television, or a musical instrument.

The CONVERTER program uses a sampling rate of approximately 1000 samples/second, but it can be changed by adjusting the counter in a delay loop in the DoSample subroutine. According to the Nyquist Sampling Theorem, the sampling rate should be at least twice the frequency of the highest-frequency component of the sound being recorded. If it is not, the sound will appear distorted when you play it back.

The CONVERTER program uses the JSL patch technique for setting up the call to the DOC read subroutine. When the subroutine is called, the DOC register number is in X and m=1, as required.

NOTE SYNTHESIZER

The free-form synthesizer used by the Sound Manager is most useful for playing back long sound patterns that have been previously digitized. It is not particularly useful for playing musical notes or synthesizing the effect of musical instruments.

It is the Note Synthesizer (tool set 25) that you will use most often to create musical melodies on the GS. It is capable of playing all standard musical notes using the characteristics of any instrument you care to define. With a minimum of extra effort, you can use the Note Synthesizer to assist the application in playing back sequences of notes in a song.

The general procedure for using the Note Synthesizer to play a pure note is quite simple:

1. Load a waveform into DOC RAM (with WriteRamBlock)

2. Get a generator (with AllocGen)

3. Play the note (with NoteOn)

4. End the note (with NoteOff)

Difficulties arise when defining the characteristics of the instrument to be used to play the note. This requires an understanding of terms such as ADSR envelope, vibrato, pitchbend, and semitone. The process of defining an instrument will be discussed later in this chapter.

Starting up the Note Synthesizer

The start-up function for the Note Synthesizer is NSStartup. Before calling it, you must call SoundStartup to start up the Sound Manager. This is necessary because the Note Synthesizer uses Sound Manager functions to perform some of its chores.

Here is how to call NSStartup:

```
PushWord UpdateRate      ;Envelope update/interrupt rate
PushPtr  MySequencer     ;Pointer to interrupt handler
_NSStartup
```

The UpdateRate variable defines an interrupt rate for the notes played by the Note Synthesizer. This is the rate at which the ADSR (attack, decay, sustain, release) envelope for the instrument selected to play the note is generated. (ADSR envelopes and instruments are described below.) The UpdateRate variable is in units of 0.4 Hz and should normally range from 75 (30 Hz) to 500 (200 Hz). The higher the rate, the better the note will sound, but the main application will be interrupted more often and will appear to slow down while music is playing.

When a Note Synthesizer update interrupt occurs, the system interrupt handler calls the user-defined MySequencer subroutine. This subroutine can keep track of the number of times it has been called and turn notes on and off at the appropriate times. For example, if the update rate is 30 Hz and a note is to play for one-half second, MySequencer would wait for 15 interrupts before turning the note off (and, in most cases, turning on the next note in the song).

Playing a Note

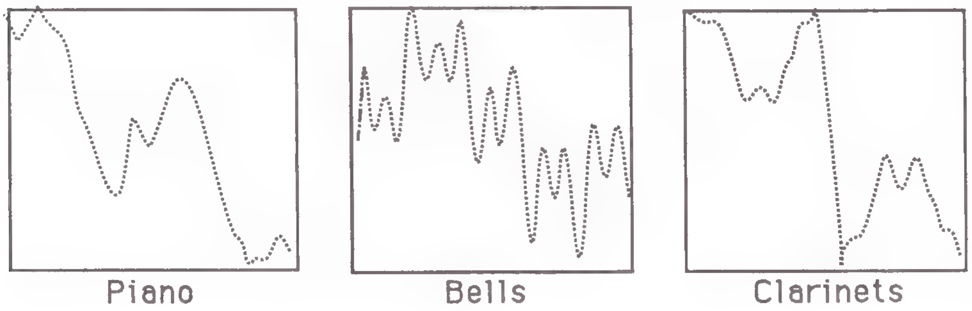
To play a note with the Note Synthesizer, you must start by using AllocGen to get a generator with which you can work:

```
PHA                      ;space for result
PushWord GenPriority     ;generator priority
_AllocGen
PLA                      ;Pop generator number
```

The GenPriority parameter passed to AllocGen is a number from 1 to 128 and represents a priority code for the generator. For most purposes, you can assign a priority of 127 to all allocated generators.

The priority indicates the degree to which the generator will be "stolen" by subsequent calls to AllocGen if all generators are in use. (Generators 0 to 13 may

Figure 11–8. Waveforms for Different Musical Instruments



be allocated; the Sound Manager and Note Synthesizer reserve one generator each.) AllocGen first tries to allocate a generator that is not in use (priority = 0). If it cannot find one, it steals one with the lowest priority that is less than or equal to GenPriority. (A generator with a priority of 128 cannot be stolen, however.) AllocGen returns an error if no generator qualifies.

If AllocGen does allocate a generator, it assigns-it a priority of GenPriority.

Once you have allocated a generator, you must load a waveform for the generator into memory. For simplicity, you can just load a simple sinusoidal wave or a triangle-shaped approximation. Keep in mind, however, that real musical instruments have different waveforms, so you should use them for more realistic sound. The waveforms for some common instruments are shown in figure 11–8.

To play the note, call NoteOn:

```
PushWord GenNum           ;generator number
PushWord Semitone         ;MIDI semitone (frequency)
PushWord Volume           ;Volume
PushPtr  MyInstrument     ;Pointer to instrument definition
_NoteOn
```

MIDI stands for Musical Instrument Digital Interface, an international standard for digital musical devices. MIDI semitone codes range from 0 to 127, where middle C is 60. Semitone codes for other notes can be deduced from the information in table 11–4.

The Volume parameter also ranges from 0 to 127 and is roughly equivalent to MIDI velocity (the speed at which a note is struck). Each group of 16 units of the volume parameter corresponds to a 6-decibel change in volume.

The MyInstrument parameter is by far the most complex parameter. It points to a parameter block that describes, in meticulous detail, changes to be made to the

Table 11–4: MIDI Semitone Codes Used by the Note Synthesizer

Octave	*Semitone Code*	*Pitch*	*Offset within Octave**
1	36	C	+0
2	48	C#	+1
3	60	D	+2
4	72	D#	+3
5	84	E	+4
		F	+5
		F#	+6
		G	+7
		G#	+8
		A	+9
		A#	+10
		B	+11

*Add these codes to the semitone base code for the octave being used.

basic waveform when playing the note. By adjusting the instrument parameters, you can simulate the characteristics of any musical instrument.

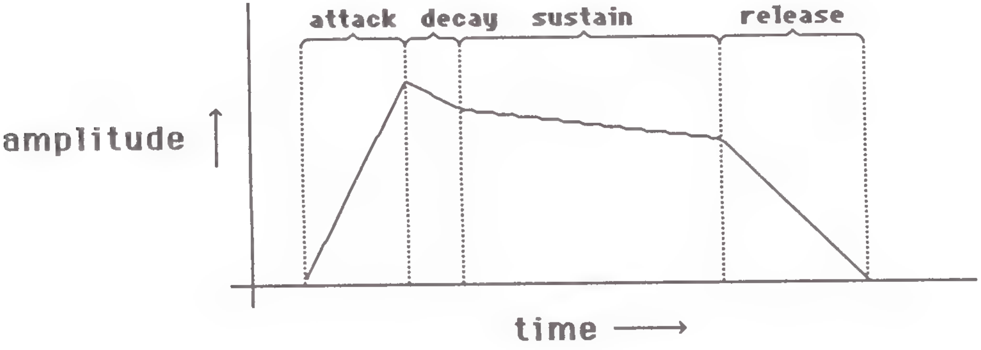
The general form of an instrument definition is shown in table 11–5. An actual example is shown in listing 11–3. It begins with the definition of an ADSR envelope for the instrument. As shown in figure 11–9, ADSR stands for attack, decay, sustain, and release, the four consecutive phases a waveform passes through when it is played. The amplitude of the ADSR envelope at any given time represents the maximum amplitude (volume) of the underlying waveform for the note; the waveform is still oscillating at the proper frequency, but its maximum amplitude is being modulated.

To explain the meaning of attack, decay, sustain, and release, it is best to consider one particular instrument, say a piano. The slope of the attack stage is proportional to how hard you strike the piano key; the harder you strike it, the faster the volume reaches its maximum. *Decay* refers to the degradation in volume after the attack maximum has been reached. *Sustain* refers to how the volume changes as you hold the key down. For a piano, the sound is sustained for some time, but gradually

Table 11–5: The Instrument Definition Data Structure

Field Name	*Offset*
ADSR_Envelope	$00
ReleaseSegment	$18
PriorityIncrement	$19
PitchBendRange	$1A
VibratoDepth	$1B
VibratoSpeed	$1C
Spare	$1D
AWaveCount	$1E
BWaveCount	$1F
AWaveList	$20
BWaveList	$20 + 6 * AWaveCount

Figure 11–9. The ADSR Envelope for a Sound



fades away; for other types of instruments, such as an organ, there is little change in volume during the sustain stage. The final stage is the release stage, which corresponds to the complete release of the key. For most instruments, the volume drops off rapidly during the release stage.

The meaning of each parameter in an instrument definition is discussed below.

ADSR_Envelope. The ADSR_Envelope field is made up of eight 3-byte entries that define the shape of the ADSR envelope at eight different stages:

- Stage 1: volume 1 (byte), increment 1 (word)
- Stage 2: volume 2 (byte), increment 2 (word)
- Stage 3: volume 3 (byte), increment 3 (word)
- Stage 4: volume 4 (byte), increment 4 (word)
- Stage 5: volume 5 (byte), increment 5 (word)
- Stage 6: volume 6 (byte), increment 6 (word)
- Stage 7: volume 7 (byte), increment 7 (word)
- Stage 8: volume 8 (byte), increment 8 (word)

In each threesome, the first byte is a volume (0 to 127) and the following word is an increment. For a given stage, the increment value indicates how fast the volume is to change from what it was in the previous stage to what it is in the current stage. That is, the increment sets the slope of the ADSR envelope for a particular time interval.

The time needed to slope up (or down) to the volume of a particular stage can be calculated from the following formula:

$$\frac{\text{time} = \text{abs(last volume} - \text{new volume)} * 256}{\text{increment} * \text{update rate}}$$

where abs means absolute value and the update rate is the one passed to NSStartup.

ReleaseSegment. The ReleaseSegment field of the instrument definition contains the number (minus one) of the ADSR_Envelope stage to which the Note Synthesizer goes when the note is released (a note is released when you call NoteOff). In most cases, the instrument will release to a zero volume in one stage, so the volume byte for this stage will be 0.

PriorityIncrement. This field contains the number that will be subtracted from the generator priority when the note reaches the sustain segment. If you allocate all generators at the same priority, this will force older notes to be stolen first, which is usually what you want to happen.

PitchBendRange. This field contains the number of MIDI semitones the pitch of the waveform will be raised when the pitchwheel reaches 127. PitchBendRange can be 1, 2, or 4.

VibratoDepth. This field contains the amplitude of a triangle-shaped wave that modulates the main waveform for the note. It can range from 0 (no vibrato) to 127.

VibratoSpeed. This field controls the frequency of the vibrato modulation and can range from 0 to 255. The exact frequency depends on the update rate passed to NSStartup.

AWaveCount. AWaveCount is a byte containing the number of wave definitions (1 to 255) in the AWaveList field for the first oscillator used by the generator.

BWaveCount. BWaveCount is a byte containing the number of wave definitions (1 to 255) in the BWaveList field for the second oscillator used by the generator.

AWaveList and BWaveList. There are two wavelists, one for each oscillator in the generator used to play a note. Using wavelists, you can assign different waveforms to different semitone ranges, change the pitch slightly, and set the DOC operating mode. All the waveforms must be loaded into DOC RAM before the note is played.

Each entry in a wavelist has the following format:

TopKey	(byte)	Semitone limit
WaveAddress	(byte)	DOC Address value
WaveSize	(byte)	DOC Waveform value
DOCMode	(byte)	DOC Oscillator Control value
RelPitch	(word)	Pitch tuning factor

TopKey. TopKey represents the highest MIDI semitone to which the waveform described by the wavelist applies. The lowest applicable semitone is the TopKey value for the previous wavelist entry plus 1. Wavelist entries must be in increasing TopKey order, and the TopKey for the last entry must be 127 (the highest MIDI semitone allowed).

WaveAddress. This value indicates the starting page of the waveform in the DOC RAM area. The Note Synthesizer puts this value directly into the DOC Address register.

WaveSize. This code represents the Address Table and Resolution for the DOC oscillator. The Note Synthesizer stores this value in the DOC Waveform register.

DOCMode. This value is placed directly in the Oscillator Control register. As mentioned earlier in this chapter, this register controls the output channel number, oscillator interrupts, operating mode, and halt/play status. When using the Note Synthesizer, you normally want the oscillator to be running in free-run mode in channel #0 with no interrupts, so the DOCMode byte is $00. It is also common to have the first oscillator run in swap mode ($06) for the attack, so that it will swap to the second oscillator operating in free-run mode for the rest of the note. In this situation, the second oscillator initially has a DOCMode of $01 (free-run, but halted); its halt bit is cleared automatically by the DOC when the swap takes place.

RelPitch. This number is used to change the pitch of a waveform in a minor way. The low-order byte is in units of 1/256 semitone; the high-order byte is a two's-complement signed number representing whole semitones. RelPitch is normally set to 0.

Turning off a Note

When it is time to turn off a note, call the NoteOff function. This forces the Note Synthesizer to go directly to the release stage in the note's ADSR envelope. For most instruments, this causes the volume of the note to quickly decay to a zero level. When this happens, no more sound is heard, the generator used for the note is automatically deallocated, and its priority is set to 0.

Here is how to call NoteOff:

```
    PushWord GenNum          ;Generator number for the note
    PushWord Semitone        ;Semitone value for the note
    _NoteOff
```

If you want to turn off all notes at once, use AllNotesOff. It requires no parameters.

The Note Synthesizer also has a function for explicitly deallocating a generator—DeallocGen:

```
    PushWord GenNum          ;generator number
    _DeallocGen
```

This function sets the priority of the generator to 0 and halts its two oscillators. You should not need to use DeallocGen, because the Note Synthesizer calls it for you automatically when the note volume goes to 0.

Shutting Down the Note Synthesizer

When you are finished with the Note Synthesizer, call NSShutDown (no parameters). It automatically turns off all generators used by the Note Synthesizer, so there is no need to call AllNotesOff or DeallocGen first.

PLAYING A SONG

The SONG program in listing 11–4 shows how to use the Note Synthesizer to play the series of notes that make up a song. Notes are defined for two distinct musical tracks, and the program combines these tracks to play the song. In the example, the same instrument and waveform is used for each track, but with a little extra effort you could assign a different instrument and waveform to each track.

Each track in the song is defined by a series of four-word note records, each of which defines the pitch of a note, its volume and duration, and the time interval between the start of the note and the start of the next note. (You can play several notes at the same time in one track by specifying a 0 interval.) The end of the track is marked by four consecutive 0 words. Here is the exact format of a note record:

- MIDI semitone: 0 to 127
- MIDI volume: 0 to 127
- note duration: in units of 1/30 second
- note interval: in units of 1/30 second

Each unit of note duration or note interval is 1/30th of a second, because the update rate passed to NSStartup is 30 Hz. If you use a different rate, you must change the duration codes accordingly.

To add another musical track to the SONG program, first increase the value of TrackMax by 1 and place a pointer to the track's note table in the track table that follows TrackMax. Finally, insert the note table for the track and terminate it with four zero words. When you reassemble the program, the new track will be played with all the others.

The portion of the SONG program that does most of the work is the Sequencer subroutine. It is called at the update rate (30 times per second) and is responsible for keeping track of the status of all notes being played and for turning notes on and off when necessary.

When Sequencer first gets control, it decrements the note interval counters for each track; these counters were originally set equal to the value stored in the record for the last note played in each track. (The program puts 1s in these counters when it first starts up to force the first notes in each track to be played right away.) If a counter becomes 0, it is time to play the next note in the track, so Sequencer calls the NextNote subroutine to get the next note record from the list. It then turns the note on by allocating a generator with AllocGen and then calling NoteOn. It also adds 8 to the track's note pointer so that it points to the next note record.

After Sequencer turns on any notes that are ready to be played, it decrements the duration counters for all generators that are still playing notes. These counters were initialized when the note was first turned on. If a counter becomes 0, it calls NoteOff to release the note. The generator is automatically deallocated when the volume goes to 0.

Table R11–1: The Major Functions in the Sound Manager Tool Set ($08)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
FFGeneratorStatus	$11	result (W)	Generator status word
		GenNum (W)	Generator number
FFSoundDoneStatus	$14	result (W)	0 = playing/$FFFF = stopped
		GenNum (W)	Generator number
FFSoundStatus	$10	result (W)	Generator status word
FFStartSound	$0E	GenMode (W)	Generator (high), mode (low)
		ParamBlock (L)	Ptr to parameter block
FFStopSound	$0F	GenMask (W)	Generator mask
GetSoundVolume	$0C	result (W)	Volume setting
		GenNum (W)	Generator number
GetTableAddress	$0B	result (L)	Ptr to low-level DOC table
ReadRamBlock	$0A	DestPtr (L)	Ptr to destination address
		DOCStart (W)	Starting address in DOC RAM
		Count (W)	Size of wave to read
SetSoundMIRQV	$12	MasterIRQV (L)	Ptr to new sound IRQ handler
SetSoundVolume	$0D	VolSetting (W)	Volume level
		GenNum (W)	Generator number
SetUserSoundIRQV	$13	result (L)	Old user IRQ handler
		NewIRQV (L)	Ptr to new user IRQ handler
SoundShutDown	$03	[no parameters]	
SoundStartup	$02	DPAddr (W)	Address of 1 page in bank 0
WriteRamBlock	$09	SourcePtr (L)	Ptr to start of data
		DOCStart (W)	Starting address in DOC RAM
		Count (W)	Size of wave to write

Table R11–2: Sound Manager Error Codes

Error Code	*Description of Error Condition*
$0810	The system does not contain a DOC chip.
$0811	The specified DOC address is invalid.
$0812	The Sound Manager has not been initialized.
$0813	The generator number is invalid.
$0814	The synthesizer mode is invalid.
$0815	The generator is busy.
$0817	The master IRQ vector has not been assigned.
$0818	The Sound Manager has already been started up.

The Sound Manager can also return Memory Manager and ProDOS 16 error codes.

Table R11–3: The Major Functions in the Note Synthesizer Tool Set ($19)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
AllNotesOff	$0D	[no parameters]	
AllocGen	$09	result (W)	Generator number allocated
		GenPriority (W)	Generator priority
DeallocGen	$0A	GenNum (W)	Generator number
NoteOff	$0C	GenNum (W)	Generator number
		Semitone (W)	MIDI semitone
NoteOn	$0B	GenNum (W)	Generator number
		Semitone (W)	MIDI semitone
		Volume (W)	Volume
		InstrumentPtr (L)	Ptr to instrument definition

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
NSShutDown	$03	[no parameters]	
NSStartup	$02	UpdateRate (W)	Update rate code
		UserUpdate (L)	Ptr to update handler

Table R11–4: Note Synthesizer Error Codes

Error Code	*Description of Error Condition*
$1901	The Note Synthesizer has already been initialized.
$1902	The Sound Manager has not been initialized.
$1921	No generators are available.
$1922	Invalid generator number.
$1923	The Note Synthesizer has not been initialized.
$1924	The generator is already being used.

Table R11–5: Useful Functions in the Miscellaneous Tool Set ($03)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
SysBeep	$2C	[no parameters]	

Listing 11–1: Creating Simple Waveforms

(a) Creating and loading a sine wave:

```
LoadSine    ANOP

            PushPtr WaveForm           ;Pointer to waveform
            PushWord #$0000            ;Starting DOC address
            PushWord #$100             ;Number of bytes
            _WriteRamBlock
            RTS

; This is a sine wave. The values were calculated from
; the formula Y = 127 * SIN(X * 2 * PI/256) + 128 where
; PI = 3.14156.

WaveForm    ANOP

DC   I1'128,131,134,137,140,144,147,150,153,156,159,162,165,168,171,174'
DC   I1'177,179,182,185,188,191,193,196,199,201,204,206,209,211,213,216'
DC   I1'218,220,222,224,226,228,230,232,234,235,237,239,240,241,243,244'
DC   I1'245,246,248,249,250,250,251,252,253,253,254,254,254,255,255,255'
DC   I1'255,255,255,255,254,254,254,253,253,252,251,250,250,249,248,246'
DC   I1'245,244,243,241,240,239,237,235,234,232,230,228,226,224,222,220'
DC   I1'218,216,213,211,209,206,204,201,199,196,193,191,188,185,182,179'
DC   I1'177,174,171,168,165,162,159,156,153,150,147,144,140,137,134,131'
DC   I1'128,125,122,119,116,112,109,106,103,100,97,94,91,88,85,82'
DC   I1'79,77,74,71,68,65,63,60,57,55,52,50,47,45,43,40'
DC   I1'38,36,34,32,30,28,26,24,22,21,19,17,16,15,13,12'
DC   I1'11,10,8,7,6,6,5,4,3,3,2,2,2,1,1,1'
DC   I1'1,1,1,1,2,2,2,3,3,4,5,6,6,7,8,10'
DC   I1'11,12,13,15,16,17,19,21,22,24,26,28,30,32,34,36'
DC   I1'38,40,43,45,47,50,52,55,57,60,63,65,68,71,74,77'
DC   I1'79,82,85,88,91,94,97,100,103,106,109,112,116,119,122,125'
```

(b) Creating and loading a triangle wave:

```
LoadTriang ANOP

            SEP      #$20              ;8-bit accumulator
            LONGA    OFF

            LDX      #0
            LDA      #$40              ;start of triangle
PutWave1    STA      WaveForm,X
            INC      A                 ;move up
            INX
            CPX      #$80
            BNE      PutWave1
```

```
PutWave2   STA      WaveForm,X
           DEC      A                  ;move down
           INX
           CPX      #$100
           BNE      PutWave2

           REP      #$20               ;16-bit accumulator
           LONGA    OFF

           PushPtr WaveForm            ;Pointer to waveform
           PushWord #$0000             ;Starting DOC address
           PushWord #$100              ;Number of bytes
           _WriteRamBlock

           RTS

WaveForm   DS       256
```

Listing 11–2: The CONVERTER Program

```
**********************************************
* This program shows how to digitize an incoming *
* analog signal through the Ensoniq DOC.         *
**********************************************

           KEEP     ANALOG
           MCOPY    ANALOG.MAC

DerefLoc   GEQU     $00                ;Used for dereferencing handles
TablePtr   GEQU     $04                ;Pointer to low-level table
Sample     GEQU     $08                ;Pointer to start of buffer

Converter  START
           USING    GlobalData

           PHK
           PLB                         ;Program bank = data bank

           JSR      StartUp            ;Start up the tools

; Allocate a 32K buffer for the samples:

           PHA                         ;Space for result (handle)
           PHA
           PushLong #$8000             ;Reserve 32K block
           PushWord MyID               ;ID tag for memory block
           PushWord #$C000             ;Locked, fixed
           PushLong #$00               ; [no meaning]
           _NewHandle
           PopLong DerefLoc
```

```
         LDA      [DerefLoc]          ;Convert handle to pointer
         STA      Sample
         LDY      #2
         LDA      [DerefLoc],Y
         STA      Sample+2

; Get address of low-level subroutine table:

         PHA                          ;space for result
         PHA
         _GetTableAddress             ;Get address
         PopLong TablePtr

; Store the 3-byte address of the DOC read subroutine (first
; entry in the table) in the operand of the JSL instruction:

         LDA      [TablePtr]
         STA      JSL_Patch+1
         LDY      #1
         LDA      [TablePtr],Y
         STA      JSL_Patch+2

         JSR      DoSample            ;Start sampling

; [insert code here to save sample to disk, etc.]

         JMP      ShutDown            ;Finish up


; This subroutine samples the analog input line until
; $8000 samples have been taken or until ESC is pressed.
; The sample size is returned in the Y register.

DoSample   ANOP

           SEP      #$20              ;8-bit accumulator
           LONGA    OFF

           LDY      #0                ;Initialize buffer pointer
GetSample  PHY                        ;Save pointer

           LDX      #$E2              ;Analog-to-Digital register

; Note: the address of the DOC read subroutine was stored in the
; three operand bytes of this JSL subroutine at the beginning of
; the program:

JSL_Patch  JSL      $123456           ;Read the register

           PLY                        ;Get pointer
           STA     [Sample],Y              ;Save the sample in buffer

           INY                             ;Bump buffer pointer
           CPY     #$8000                  ;At 32K limit?
           BEQ     Exit                    ;Yes, so branch

 ; Use a delay loop like this to adjust the sampling rate.
 ; A value of 475 for X gives a rate of about 1000 samples/second.

           LDX     #475                    ;Delay counter
Delay      DEX
           BNE     Delay

 ; Check keyboard for ESC (see chapter 12):

           LDA     $E0C000                 ;Check keyboard
           BPL     GetSample               ;Branch if nothing there

           STA     $E0C010                 ;Clear keyboard strobe

           CMP     #$9B                    ;Esc?
           BNE     GetSample               ;No, so get next sample

Exit       REP     #$20                    ;Back to 16-bit accumulator
           LONGA   ON
           RTS

           END

***********************************
* Start up the standard tool sets *
***********************************
StartUp    START
           USING GlobalData

           _TLStartup
           _MTStartup

           PHA
           _MMStartup
           PLA
           STA     MyID

* Get one page in bank $00 for SoundStartup:

           PHA                             ;Space for result (handle)
           PHA
           PushLong #$100                  ;Reserve one page
           PushWord MyID                   ;ID tag for memory block
           PushWord #$C005                 ;Locked, aligned, fixed
           PushLong #$00                   ;... in bank $00
           _NewHandle
```

```
; Dereference the handle:

            PopLong DerefLoc
            LDA     [DerefLoc]          ;Get direct page address

            PHA                         ;Push direct page address
            _SoundStartup               ;Start up the Sound Manager

            RTS

; Shut down all the tool sets
; and leave the application:

ShutDown     ENTRY

            _SoundShutDown

            PushWord MyID
            _MMShutDown

            _MTShutDown
            _TLShutDown

            _Quit   QuitParms

QuitParms   DC      I4'0'
            DC      I2'0'

            END


GlobalData  DATA

MyID        DS      2                   ;program ID tag

            END
```

Listing 11–3: An Example of an Instrument Definition

```
Instrument ANOP
           DC     I1'127,0,127'   ;Sharp attack to max volume
           DC     I1'120,20,1'    ;Slow decay
           DC     I1'120,0,0'     ;Sustain at same level
           DC     I1'0,0,$10'     ;Release to zero
           DC     I1'0,0,0'       ;[unused stage]
           DC     I1'0,0,0'       ;[unused stage]
           DC     I1'0,0,0'       ;[unused stage]
           DC     I1'0,0,0'       ;[unused stage]
           DC     I1'3'           ;Release stage number
           DC     I1'32'          ;Priority reduction at sustain
           DC     I1'2'           ;Pitchbend
           DC     I1'80'            ;Vibrato depth
           DC     I1'90'            ;Vibrato frequency
           DC     I1'0'             ;[spare]
           DC     I1'1'             ;Number of entries in AWavelist
           DC     I1'1'             ;Number of entries in BWavelist
AWaveList  DC     I1'127'           ;TopKey
           DC     I1'$00'           ;Starts at page $00 in DOC RAM
           DC     I1'0'             ;Waveform register: one page
           DC     I1'6'             ;Mode: swap to B, playing
           DC     I2'0'             ;No pitch change
BWaveList  DC     I1'127'           ;TopKey
           DC     I1'$01'           ;Starts at page $01 in DOC RAM
           DC     I1'0'             ;Waveform register: one page
           DC     I1'1'             ;Mode: free-run, halted
           DC     I2'0'             ;No pitch change
```

Listing 11–4: The SONG Program

```
**************************************************
*  This program shows how to construct a simple *
*  tune using the Note Synthesizer.              *
**************************************************

           KEEP    SONG
           MCOPY   SONG.MAC

NotePtr    GEQU    $0                  ;Pointers to each note track
Interval   GEQU    $20                 ;time until next note begins

Song       START
           USING   GlobalData
           USING   TheNotes
           USING   Instrument

           PHK
           PLB                         ;Program bank = data bank

           TDC
           STA     MyDP                ;Save current direct page

           JSR     StartUp

           JSR     MakeWave            ;Load wave into DOC RAM area

; Get the Note Synthesizer up and running:

           SEI                         ;No interrupts until we're ready

           PushWord #75                ;30 Hz update rate
           PushPtr  Sequencer          ;Interrupt handler
           _NSStartup
```

```
; Here is the main program loop:

KeyWait    PRINTLN 'Press any key to start the song (ESC to cancel):'

           PHA
           PushWord #0              ;0 = no echo
           _ReadChar
           PLA

           AND     #$7F             ;Convert to ASCII
           CMP     #$1B             ;ESC pressed?
           BEQ     ShutDown         ;Yes, so bail out

; Initialize the pointers to the sound tables and fix
; things up so that the first note in each track is
; played right away:

           LDY     #0
           LDX     #0
DoTables   LDA     TrackMax+2,X     ;Transfer track pointer to d.p.
           STA     NotePtr,X
           LDA     TrackMax+4,X
           STA     NotePtr+2,X

           LDA     #1
           STA     Interval,X       ;(Causes 1st note to be played)

           INX                      ;Move to next entry
           INX
           INX
           INX

           INY
           CPY     TrackMax         ;Done all tracks?
           BNE     DoTables         ;No, so branch

           CLI                      ;Allow synthesizer interrupts!

           BRL     KeyWait


; Shut down all the tool sets
; and leave the application:

ShutDown   _NSShutDown
           _TextShutDown
           _SoundShutDown
           PushWord MyID
           _MMShutDown

           _MTShutDown
           _TLShutDown

           _Quit    QuitParms

QuitParms  DC       I4'0'
           DC       I2'0'

           END

***********************************************************
*  This subroutine gets control after a Note Synthesizer *
*  interrupt. Interrupts occur 30 times/second.          *
***********************************************************
Sequencer  START
           USING Instrument
           USING GlobalData
           USING TheNotes

           PHB
           PHK
           PLB                          ;Allow absolute addressing

           PHD
           LDA      MyDP                ;Switch to application's d.p.
           TCD

; Check to see if it's time to play the next note:

           LDX      #0
DoTrack    DEC      Interval            ;Reduce time to next note
           BNE      NextTrack           ;Branch if not time to play next note

           PHX
           JSR      NextNote            ;Get next note
           PLX

NextTrack  TDC
           CLC
           ADC      #4                  ;Keep the d.p. in step with
           TCD                          ; the current track.

           INX
           CPX      TrackMax            ;Done all tracks?
           BNE      DoTrack             ;No, so branch
```

```
; Turn off all generators whose notes are done:

           LDX       #13*2
GenCheck   LDA       Duration,X          ;Is this note playing?
           BEQ       NextGen             ;No, so branch

           DEC       Duration,X          ;Reduce duration counter
           BNE       NextGen             ;Branch if more to come

           PHX                           ;Don't destroy X

           LDA       Generator,X
           PHA
           LDA       Semitone,X
           PHA
           _NoteOff                      ;Turn off the note

           PLX                           ;Restore X

NextGen    DEX                           ;Move to next generator
           DEX
           BPL       GenCheck

           PLD                           ;Restore direct page
           PLB                           ;Restore data bank
           RTL

; Get the next note in the note table:

NextNote   LDA       [NotePtr]           ;Get next semitone
           BNE       GetGen              ;Branch if note defined
           RTS

GetGen     PHA                           ;Space for result
           PushWord #127                 ;Generator priority
           _AllocGen                     ;Allocate a generator
           PLA                           ;Get generator number
           PHA                           ;(keep a copy on stack)
           PHA                           ;(... and another)

           ASL       A                   ;x2 for array position
           TAX

           PLA
           STA       Generator,X         ;Store generator number

           LDA       [NotePtr]
           STA       Semitone,X          ;Store note semitone

           LDY       #4
           LDA       [NotePtr],Y
           STA       Duration,X          ;Store note duration
           LDY      #6
           LDA      [NotePtr],Y
           STA      Interval            ;Store time to next note

; Sound the note by calling NoteOn (the generator
; number is already on stack):

           LDA      [NotePtr]
           PHA                          ;Semitone
           LDY      #2
           LDA      [NotePtr],Y
           PHA                          ;Volume
           PushPtr  Instrument          ;Instrument
           _NoteOn                      ;Start sounding the note

* Bump the pointer to the next note record:

           CLC
           LDA      NotePtr
           ADC      #8                  ;Add size of note record
           STA      NotePtr

           LDA      Interval            ;Delay before next note?
           BEQ      NextNote            ;No, so get next note right now

           RTS

Generator  DS       14*2                ;Generators used by active notes
Semitone   DS       14*2                ;Semitones used by active notes
Duration   DS       14*2                ;Durations of active notes

           END


***************************************
* This data segment defines the notes *
* for the tune we will play.          *
***************************************
TheNotes   DATA

; These are the MIDI codes for semitones:

o5         EQU      84                  ;Octave 5 base
o6         EQU      96                  ;Octave 6 base

; Semitone offsets from the start of each octave:

C          EQU      0
Cs         EQU      1
D          EQU      2
Ds         EQU      3
```

```
E            EQU      4
F            EQU      5
Fs           EQU      6
G            EQU      7
Gs           EQU      8
A            EQU      9
As           EQU      10
B            EQU      11

;This a table of pointers to each music track:

TrackMax     DC       I2'2'               ;Number of tracks
             DC       I4'Track1'          ;Pointer to track 1
             DC       I4'Track2'          ;Pointer to track 2

; Here are the note definitions. Each is made up
; of four words describing the semitone, volume,
; duration, and the interval until the next note.
; A zero semitone marks the end of the list.

; This is the definition for "Happy Birthday":

Track1       ANOP

v1           EQU      127                 ;Melody volume

             DC       I'o5+C,v1,4,4'
             DC       I'o5+C,v1,4,4'
             DC       I'o5+D,v1,8,8'
             DC       I'o5+C,v1,8,8'
             DC       I'o5+F,v1,8,8'
             DC       I'o5+E,v1,15,15'

             DC       I'o5+C,v1,4,4'
             DC       I'o5+C,v1,4,4'
             DC       I'o5+D,v1,8,8'
             DC       I'o5+C,v1,8,8'
             DC       I'o5+G,v1,8,8'
             DC       I'o5+F,v1,15,15'

             DC       I'o5+C,v1,4,4'
             DC       I'o5+C,v1,4,4'
             DC       I'o6+C,v1,8,8'
             DC       I'o5+A,v1,8,8'
             DC       I'o5+F,v1,8,8'
             DC       I'o5+E,v1,8,8'
             DC       I'o5+D,v1,15,15'

             DC       I'o5+As,v1,4,4'
             DC       I'o5+As,v1,4,4'
             DC       I'o5+A,v1,8,8'
```

```
         DC       I'o5+F,v1,8,8'
         DC       I'o5+G,v1,8,8'
         DC       I'o5+F,v1,15,15'

         DC       I'0,0,0,0'      ;End of note table

; Track2 contains the accompanying chords:

Track2   ANOP

v2       EQU      127                ;Chord volume

         DC       I'o5+F,0,8,8'      ;(do nothing)

         DC       I'o5+F,v2,24,0'
         DC       I'o5+A,v2,24,0'
         DC       I'o6+C,v2,24,24'

         DC       I'o5+E,v2,23,0'
         DC       I'o5+As,v2,23,0'
         DC       I'o6+C,v2,23,23'

         DC       I'o5+F,v2,16,0'
         DC       I'o5+A,v2,16,0'
         DC       I'o6+C,v2,16,16'

         DC       I'o5+E,v2,8,0'
         DC       I'o5+As,v2,8,0'
         DC       I'o6+C,v2,8,8'

         DC       I'o5+F,v2,23,0'
         DC       I'o5+A,v2,23,0'
         DC       I'o6+C,v2,23,23'

         DC       I'o5+F,v2,24,0'
         DC       I'o5+A,v2,24,0'
         DC       I'o6+C,v2,24,24'

         DC       I'o5+E,v2,31,0'
         DC       I'o5+As,v2,31,0'
         DC       I'o6+C,v2,31,31'

         DC       I'o5+F,v2,16,0'
         DC       I'o5+A,v2,16,0'
         DC       I'o6+C,v2,16,16'

         DC       I'o5+E,v2,8,0'
         DC       I'o5+As,v2,8,0'
         DC       I'o6+C,v2,8,8'
```

```
          DC      I'o5+F,v2,15,0'
          DC      I'o5+A,v2,15,0'
          DC      I'o6+C,v2,15,15'

          DC      I'0,0,0,0'      ;End of note table

          END


****************************************
* This data defines the instrument which *
* is used to play the song.               *
****************************************
Instrument DATA

          dc      i1'$7F,0,64'     ;Sharp attack
          dc      i1'0,0,$02'      ;Sustain with slow decay
          dc      i1'0,0,0'        ; (the other six segments
          dc      i1'0,0,0'        ;   are unused)
          dc      i1'0,0,0'
          dc      i1'0,0,0'
          dc      i1'0,0,0'
          dc      i1'0,0,0'

          dc      i1'1'            ;Segment number for release
          dc      i1'32'           ;Generator priority
          dc      i1'0'            ;Pitch bend range
          dc      i1'0'            ;Vibrato
          dc      i1'0'            ;Vibrato speed
          dc      i1'0'            ; [unused]

          dc      i1'1'            ;Number of waveforms for osc #1
          dc      i1'1'            ;Number of waveforms for osc #2
Wave1     dc      i1'127,0,0,0,0,0'
Wave2     dc      i1'127,0,0,0,0,0'

          END


*************************************************
* Load a waveform into the first page of DOC RAM *
*************************************************
MakeWave   START

; Transfer the waveform to the DOC RAM:

          PushPtr  WaveForm    ;Pointer to the wave
          PushWord #$0000      ;Starting address in DOC RAM
          PushWord #$100       ;Transfer one page
          _WriteRamBlock       ;Do the transfer!
          RTS
```

```
; This is a sine wave. The values were calculated from
; the formula Y = 64*SIN(X * 2*PI/256) + 128 where
; PI = 3.14156. The zero baseline is given by the value $80.

WaveForm   ANOP

DC  I1'128,130,131,133,134,136,137,139,140,142,144,145,147,148,150,151'
DC  I1'152,154,155,157,158,160,161,162,164,165,166,167,169,170,171,172'
DC  I1'173,174,175,176,177,178,179,180,181,182,183,184,184,185,186,187'
DC  I1'187,188,188,189,189,190,190,190,191,191,191,192,192,192,192,192'
DC  I1'192,192,192,192,192,192,191,191,191,190,190,190,189,189,188,188'
DC  I1'187,187,186,185,184,184,183,182,181,180,179,178,177,176,175,174'
DC  I1'173,172,171,170,169,167,166,165,164,162,161,160,158,157,155,154'
DC  I1'152,151,150,148,147,145,144,142,140,139,137,136,134,133,131,130'
DC  I1'128,126,125,123,122,120,119,117,116,114,112,111,109,108,106,105'
DC  I1'104,102,101,99,98,96,95,94,92,91,90,89,87,86,85,84'
DC  I1'83,82,81,80,79,78,77,76,75,74,73,72,72,71,70,69'
DC  I1'69,68,68,67,67,66,66,66,65,65,65,64,64,64,64,64'
DC  I1'64,64,64,64,64,64,65,65,65,66,66,66,67,67,68,68'
DC  I1'69,69,70,71,72,72,73,74,75,76,77,78,79,80,81,82'
DC  I1'83,84,85,86,87,89,90,91,92,94,95,96,98,99,101,102'
DC  I1'104,105,106,108,109,111,112,114,116,117,119,120,122,123,125,126'

           END


**********************************
* Start up the standard tool sets *
**********************************
StartUp    START
           USING GlobalData

DeRefLoc   EQU    $0                    ;Used for dereferencing handle

           _TLStartup
           _MTStartup

           PHA
           _MMStartup
           PLA
           STA     MyID

           PushPtr ToolTable
           _LoadTools                   ;Load the Note Sequencer

* Get one page in bank $00 for SoundStartup:

           PHA                          ;Space for result (handle)
           PHA
           PushLong #$100               ;Reserve one page
           PushWord MyID                ;ID tag for memory block
```

```
          PushWord #$C005          ;Attributes -- locked, aligned, fixed
          PushLong #$00            ;... in bank $00
          _NewHandle

; Dereference the handle:

          PLA
          STA     DerefLoc
          PLA
          STA     DerefLoc+2
          LDA     [DerefLoc]       ;Get direct page address

          PHA                      ;Push direct page address

          SoundStartup             ;Start up the Sound Manager

          _TextStartup

          RTS

ToolTable DC      I2'1'            ;One tool set to load
          DC      I2'25,$0000'     ;Note Sequencer tool set

          END

GlobalData DATA

MyDP      DS      2                ;program's direct page
MyID      DS      2                ;program ID tag

          END
```

CHAPTER 12

Using the Text Tool Set

The purpose of the Text Tool Set (tool set 12) is to give programs running in 65816 native mode anywhere in memory an easy way to direct character output or input operations to an I/O device in a slot or port. Doing this without the Text Tool Set is awkward, because these types of I/O devices must be accessed while in emulation mode and while the data bank and direct page registers are set to 0. (These restrictions stem from the need to maintain compatibility with IIe software and hardware.) The functions in the Text Tool Set take care of saving the current mode and registers, switching you to the proper ones for the I/O operation, and restoring the original values on exit.

The three main uses of the Text Tool Set are to read character input from the keyboard (without using the Event Manager), to display characters on the GS's 80-column text screen (instead of the super high-resolution screen used by QuickDraw II), and to send data to a printer. Although these uses will be emphasized in this chapter, the Text Tool Set is general enough to work with any character device in any slot or port, or with custom RAM-based I/O drivers.

The start-up and shut-down functions for the Text Tool Set are TextStartup and TextShutDown. Neither one requires parameters or returns results.

LOGICAL DEVICES AND MASKS

The functions in the Text Tool Set deal with three logical I/O devices:

- An input device
- An output device
- An error-output device

The error-output device is the device to which an application can send error messages or status reports without interfering with what the primary output device receives.

You can assign each logical device to any slot or port on the GS that interfaces with a character-based I/O device. In a typical hardware configuration, the choices for output devices are a printer in port #1, a modem in port #2, and an 80-column video screen in port #3. The input device is usually the keyboard (it is supported through the video screen port #3) or the modem in port #2. The default device assignment for all three devices is port #3.

Device Driver Types

You can assign a specific type of device driver to each of the three logical devices with Text Tool Set functions. The three choices are:

- BASIC driver (driver type code 0)
- Pascal 1.1 driver (driver type code 1)
- RAM-based driver (driver type code 2)

A BASIC driver is one which adheres to Apple's Applesoft protocol for initialization, input, and output. According to this protocol, the entry points for the driver must be as follows:

- $Cn00 (initialization)
- $Cn05 (input; character returned in accumulator)
- $Cn07 (output; character in accumulator)

where *n* is the port (or slot) number. The GS's two serial ports and the 80-column video port support this protocol.

A Pascal 1.1 driver uses a different protocol, first developed for the Apple UCSD Pascal operating system. Such drivers contain a look-up table at $Cn0D to $Cn13 that contains the low-order bytes of the addresses of the subroutines for handling initialization, input, output, status, device control, and device interrupts. The high-order bytes are always $Cn. Any driver with a value of $38 at $Cn05 and a value of $18 at location $Cn07 supports the Pascal 1.1 protocol. This includes the serial ports and video port on the GS.

You can also use your own RAM-based drivers if you wish. Such drivers must support five fundamental subroutines: initialization, input, output, status, and control. The addresses of these subroutines must appear in a table at the beginning of the driver. Each address is a 3-byte absolute address (low-order bytes first, as usual).

The GS calls RAM-based driver entry points with a JSL instruction while in full native mode, so driver subroutines must end with an RTL instruction. Character transfers between the tool function and the driver use the low-order 8 bits of the accumulator.

Because the GS internal character I/O ports support both the BASIC and Pascal protocols, you can assign a driver type code of 0 or 1 to the logical devices. The default driver type is BASIC, although the default is only set up when ProDOS 16 first starts up. There is no way for an application to tell what the driver type will be when it gets control, so the application must always explicitly set the driver type using the techniques described below.

Data Masks

When handling I/O operations involving ASCII-encoded text, it is important to set the high-order bit (bit 7) properly. The ASCII standard does not use this bit, so many peripheral devices expect it to be 0. Most printers, for example, will want the high bit off unless you are trying to print special symbols that are assigned to codes above $7F. The GS BASIC 80-column text screen driver, on the other hand, insists on receiving characters with the high-order bit set, and keyboard input subroutines on Apple II computers traditionally return characters with the high-order bit set.

The Text Tool Set always manipulates the bits in a byte you send or receive by logically ANDing the byte with an AND mask and then logically ORing the result with an OR mask (unless you are using a RAM-based driver). There are AND and OR masks for each of the three logical devices.

On initialization, the AND masks for the input, output, and error-output devices are set to $FF and the OR masks are set to $80. This forces the high-order bit of incoming and outgoing data to 1 and is the format expected by BASIC, the default driver type.

When sending data to a printer, you usually want to clear the high-order bit of outgoing data to 0. To do this, change the output AND mask to $7F and the OR mask to $00 with the SetOutGlobals function:

```
    PushWord #$7F              ;The new AND mask
    PushWord #$00              ;The new OR mask
    _SetOutGlobals             ;(or SetIn, SetErr)
```

(Use SetInGlobals and SetErrGlobals for the other two logical devices.)

When using an ImageWriter II printer, the state of the high-order bit is usually unimportant, because the ImageWriter II normally forces the high-order bit of incoming data to 0. If, however, you specifically enable recognition of this bit (with a special printer command: $1B $5A $00 $20), perhaps to allow printing of MouseText icons that have codes from $C0 to $DF, you do not want the high-order bit to be altered. In this situation, you would change the masks to $FF (AND) and $00 (OR) and use an AND #$7F instruction to specifically clear the high-order bit of a text character before printing it.

To determine the active AND and OR masks, use the GetInGlobals (input device), GetOutGlobals (output device), and GetErrGlobals (error output device) functions:

```
    PHA                     ;space for AND mask
    PHA                     ;space for OR mask
    _GetOutGlobals          ;(or GetIn, GetErr)
    PLA                     ;Pop the OR mask (low byte)
    PLX                     ;Pop the AND mask (low byte)
```

You might use these functions to save the existing masks so that you can restore them when you exit your program.

CHANGING ACTIVE DEVICES

Before you begin performing character I/O operations, you must select your three active devices, initialize them, and select appropriate AND and OR masks. To select a device, use the SetInputDevice, SetOutputDevice, and SetErrorDevice functions. Each requires two parameters: a device-type code word and a long word containing the port or slot number of the device or the address of its RAM-based driver.

For example, here is how to select the printer attached to port #1 as the active output device:

```
    PushWord #0             ;0 = BASIC driver type
    PushLong #1             ;port 1
    _SetOutputDevice
```

The first parameter pushed on the stack is the driver type code (use 0 for a BASIC driver); the second is the port number. If you were using a RAM-based driver with the printer, the second parameter would be the address of its subroutine address table.

If you need to know what the currently active device is, use the GetInputDevice, GetOutputDevice, and GetErrorDevice functions:

```
    PHA                     ;Space for device type
    PHA                     ;Space for slot number (long)
    PHA
    _GetOutputDevice        ;(or GetInput, GetError)
    PopLong OutputPort      ;Pop result
    PopWord DeviceType      ;Pop result
```

These functions each return a device type code (word) and a slot number or a pointer to a RAM-based driver (long word).

Initialization

Once you have assigned an active device for input, output, or error output, be sure to initialize it with InitTextDev. If you do not do so, it probably will not work properly. To use InitTextDev, pass it a word describing which device you wish to initialize:

```
PushWord TheDevice        ;Device to initialize
_InitTextDev
```

The permitted values for TheDevice are 0 (input device), 1 (output device), and 2 (error-output device).

The effect of initialization varies from device to device. When you initialize the port #3 output device (the video screen) for output, for example, the screen clears.

The subroutines shown in listing 12–1 illustrate how you might enable video port #3 for output, a printer in port #1 for output, and the keyboard for input.

SENDING CHARACTERS TO THE OUTPUT DEVICE

There are five different ways you can send a character, or a group of characters, to the standard output device. The method you will use depends on how the textual information is arranged in memory.

The five techniques are summarized below:

WriteChar	Send a single character
WriteLine	Send a sequence of characters that is preceded by a length byte; then send a Carriage Return code
WriteString	Send a sequence of characters that is preceded by a length byte
TextWriteBlock	Send a specified sequence of characters
WriteCString	Send a sequence of characters that is terminated by a $00 byte

(Similar functions exist for the error-output device: ErrWriteChar, ErrWriteLine, ErrWriteString, ErrWriteBlock, and ErrWriteCString.)

All these methods process outgoing characters by first ANDing them with the AND mask and then ORing them with the OR mask.

Single characters are best handled by WriteChar:

```
PushWord #$008D  ;Push with character in low byte
_WriteChar
```

To send a group of characters, you could make multiple calls to WriteChar, but this is slow and inefficient. If the text is in Pascal string format (that is, if it is preceded by a length byte), use WriteString instead:

```
          PushPtr  MyString            ;Pointer to string
          _WriteString
          RTS

MyString  STR 'This is a string'       ;Text of string
```

Use WriteLine in the same way if you want a carriage return code sent after the string is sent.

If the text is in C-string format (that is, if it is followed by a zero byte, with no length byte), use WriteCString instead of WriteString. There is no C equivalent to WriteLine, however.

Displaying text strings is such a common event that it is convenient to use macros to handle it. The PRINT macro in listing 12–2 sends a string specified in its argument to the output device. In a program, invoke PRINT by placing the string in single quote marks after the macro name:

```
    PRINT 'Print this'
```

If the string includes a single quote mark as part of the string, specify two single quote marks in a row.

The PRINTLN macro, also shown in listing 12-2, is similar to PRINT but it uses WriteLine instead of WriteString. As a result, it sends a carriage return code after the specified text string.

To send any sequence of characters inside a large block of text, use TextWriteBlock:

```
    PushPtr TheText           ;Pointer to start of text block
    PushWord Offset           ;Character to begin with
    PushWord Count            ;Number of characters to send
    _TextWriteBlock
```

TextWriteBlock sends the "Count" characters beginning at a position "Offset" bytes from the start of the text block pointed to by TheText.

READING CHARACTERS FROM THE INPUT DEVICE

There are three basic character reading functions in the Text tool set: ReadChar, ReadLine, and TextReadBlock. The one you will use most often is ReadChar, because it grabs one character at a time from the keyboard (assuming port #3 is the input port), which is how most programs prefer dealing with keyboard input. Here is how to use it:

```
    PHA                       ;space for result
    PushWord EchoFlag         ;1 - echo, 0 = don't echo
    _ReadChar
    PLA                       ;Pop character (low byte)
```

The EchoFlag indicates whether you want to echo incoming data to the output device. In most situations, you will set it to 1 to enable echoing.

ReadLine keeps reading characters until it receives an end-of-line (EOL) character, which is usually a carriage return code ($8D), or until a specified number of characters have been received. On exit, it returns the number of characters received. Here is how to call ReadLine:

```
    PHA                          ;space for result
    PushPtr  InputBuffer         ;pointer to input buffer
    PushWord MaxCount            ;size of buffer
    PushWord #$8D                ;EOL character
    PushWord EchoFlag            ;1 = echo, 0 = don't echo
    _ReadLine
    PLA                          ;Pop character count
```

InputBuffer is the area of memory in which the incoming characters are stored. MaxCount is the maximum line length, which must be less than the buffer size.

The last character-reading function is TextReadBlock. With it, you can read in a group of characters and place them at a specified location inside a text buffer:

```
    PushPtr TextArea             ;Pointer to start of text area
    PushWord Offset              ;Offset to start of input block
    PushWord Count               ;Number of characters to read
    PushWord EchoFlag            ;1 = echo, 0 = don't echo
    _TextReadBlock
```

The problem with using TextReadBlock is that it does not exit until it receives Count characters. ReadLine, on the other hand, always exits when it receives the EOL character.

KEYBOARD INPUT

One function the Text Tool Set cannot perform is to check whether or not a key has been entered from the keyboard. You will want to do this, for example, if you design a program loop which is to continue looping until the user presses a key. You cannot use ReadChar because it waits until a key has been pressed before returning control to the application. One solution is to access the keyboard I/O locations directly. (If the Event Manager is active, you can keep looping until GetNextEvent or TaskMaster indicates that a key-down event occurred.)

The two main keyboard I/O locations are as follows:

• $E0C000	keyboard strobe (bit 7) + data
• $E0C010	clear keyboard strobe

When you are accessing these locations (or any other I/O locations for that matter), the *m* status flag must be set to 1 so that read or write operations affect the one-byte I/O location only. This is very important.

To determine whether or not a key has been pressed, check the high-order bit of $E0C000 (the keyboard strobe bit). If it is 1, a key has been pressed and the lower seven bits of $E0C000 contain the key's ASCII character code. To clear the keyboard strobe so that the next read of $E0C000 will not return the same keycode, access the $E0C010 I/O location.

Here is a short subroutine you could call to check the status of the keyboard:

```
CheckKbd  SEP   #$20       ;8-bit accumulator
          LONGA OFF

          LDA   $E0C000    ;Check keyboard
          BPL   KbdExit    ;Branch if nothing there
          STA   $E0C010    ;Clear strobe

KbdExit   REP   #$20       ;16-bit accumulator
          LONGA ON
          RTS
```

If a key has been pressed, the negative flag is set to 1 and the character code is returned in the accumulator. If no key has been pressed, the negative flag is cleared to 0.

You must be careful while accessing keyboard I/O locations when keyboard interrupts are enabled, because all keypresses will be processed by an interrupt handler before you have a chance to read $E0C000 to determine whether or not a key has been pressed. (Remember, keyboard interrupts are automatically enabled if the Event Manager is in use.) This is a particularly important consideration when developing desk accessories because they may be activated at any time from any application.

You can check whether keyboard interrupts are enabled by calling the GetIRQEnable function, in the Miscellaneous tool set:

```
PHA             ;space for result
_GetIRQEnable
PLA             ;Get status word
```

If keyboard interrupts are enabled, bit 7 of the status word result will be set to 1.

When interrupts are enabled, you must first disable keyboard interrupts with IntSource, another function in the Miscellaneous tool set. Here is how to disable them:

```
PushWord #1     ;1 = disable keyboard interrupts
_IntSource
```

Later in the program you can re-enable keyboard interrupts by passing a $00 parameter to IntSource, but only do this if interrupts were enabled in the first place.

VIDEO OUTPUT

Applications using the 80-column text screen cannot take advantage of tool sets like the Window Manager, Menu Manager, and Dialog Manager because these tool sets work with the super high-resolution screen only. Thus, you have to do a lot more work if you wish to follow desktop interface guidelines. Nevertheless, text-based applications are popular alternatives to similar graphics-based ones, because screen operations such as scrolling and updating take place more quickly.

The drivers for most intelligent output devices support sets of commands you can use to control the way the devices handle subsequent characters. In most cases, a command is either a control character (a character with an ASCII code from 0 to 31) or a control character followed by a series of characters in a certain order.

The driver for the 80-column column text screen has commands that (with one exception) are single control characters. You can use these commands to perform such useful actions as screen clearing; moving the cursor position up, down, left, or right; or highlighting subsequent characters. The commands supported by the BASIC and Pascal drivers for the 80-column text screen are summarized in table 12–1.

Notice that the ASCII codes given in table 12–1 have the high bit set because the 80-column video driver expects characters in this format. If the output OR mask is $80 (the usual case), you can also send codes with the high bit off.

Also notice that for the Pascal driver, the CR (carriage return) code does not move the cursor to the next line as it does when the BASIC driver is being used. You must follow the CR with LF (line feed) to move the cursor down one line.

Character Display

Table 12–2 shows which characters appear on the text screen when you send ASCII codes from $A0 to $FF to the 80-column video driver. These characters are usually displayed in normal video (white characters on a black background) but this mode can be inverted by sending a $8F (INVERSE) code to the driver first. To return to normal video, send a $8E (NORMAL) code.

MouseText

MouseText is a group of thirty-two icons representing such objects as apples, arrows, checkmarks, a file folder, scroll bar arrows, and shading patterns. You can use them to add a little spice to your screen displays.

Each MouseText icon corresponds to a standard keyboard character, as shown in table 12–2. To display a MouseText character, first enable the driver's handling of MouseText by sending a $9B code (MTXT_ON) and turn on inverse video by sending a $8F code (INVERSE). Then send the code for the character corresponding to the

Table 12–1: Control Commands Supported by the 80-Column Screen Driver

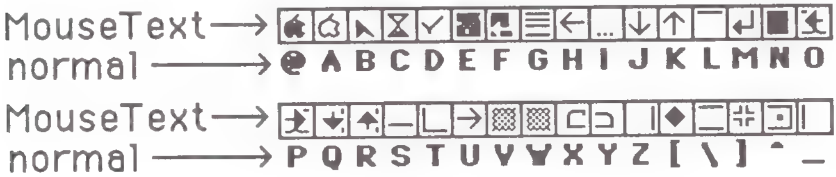
ASCII Code	*Symbolic Name*	*Action*
$85	CURSOROFF	Do not display a cursor
$86	CURSORON	Display a cursor
$87	BEEP	Sound a tone; volume, pitch set by Control Panel
$88	BACKSPACE	Move the cursor left one position
$8A	LF	Line feed; move the cursor down one line
$8B	CLREOP	Clear the screen from the current cursor position to the end of the screen
$8C	CLRSCREEN	Clear the screen; move the cursor to the top left-hand corner
$8D	CR	Carriage return; for BASIC, move the cursor to the left side of the next line; for Pascal, move the cursor to the left side of the current line (follow CR with LF to move the cursor down one line)
$8E	NORMAL	Enable the normal mode for text display
$8F	INVERSE	Enable the inverse mode for text display
$91	SET_40COL	Enable the 40-column screen display
$92	SET_80COL	Enable the 80-column screen display
$93	PAUSE	Stop receiving characters until a key is pressed
$95	FIRM_OFF	Turn off the 80-column display and firmware
$96	SCROLLD	Scroll the display down one line; leave the cursor at the same position
$97	SCROLLU	Scroll the display up one line; leave the cursor at the same position
$98	MTXT_OFF	Show inverse upper-case characters instead of MouseText characters
$99	HOME	Move the cursor to the home position (the top left-hand corner of the screen)
$9A	CLRLINE	Clear the entire line on which the cursor is positioned
$9B	MTXT_ON	Show MouseText characters instead of inverse upper-case characters

Table 12–1: Continued

ASCII Code	*Symbolic Name*	*Action*
$9C	MOVERIGHT	Move the cursor one position to the right, moving to the next line if necessary
$9D	CLREOL	Clear the screen from the current cursor position to the end of the line
$9E	SET_CURS	*For the BASIC driver only:* Change the cursor character to the character which follows
$9E	GOTOXY	*For the Pascal driver only:* Move the cursor to the position indicated by the two bytes that follow; the first byte is the horizontal position plus 32; the second is the vertical position plus 32
$9F	MOVEUP	Move the cursor up one line (in the same column); do not move if the cursor is in the top line

MouseText icon. Code $C1 (A), for example, represents the Open-Apple icon. If you want the driver to display inverse characters instead of icons, send code $98 (MTXT_OFF) to disable the conversion to MouseText icons.

These are the MouseText symbols and their corresponding characters (they correspond to the codes from $C0 to $DF):


MouseText
normal
@ A B C D E F G H I J K L M N O
MouseText
normal
P Q R S T U V W X Y Z [\ ] ^ _

Cursors

A cursor marks the position on the screen at which the next displayable character you send to the video driver will appear. It is normally a solid white box, formed by displaying an inverse space character.

If you are using the BASIC video driver, you can change the cursor to any other character by sending a $9E (SET_CURS) command to the video driver, followed by

Table 12–2: Symbols for the Displayable ASCII Character Codes

ASCII Code	*Symbol*	*ASCII Code*	*Symbol*	*ASCII Code*	*Symbol*
$A0	[space]	$C0	@	$E0	±
$A1	!	$C1	A	$E1	a
$A2	”	$C2	B	$E2	b
$A3	#	$C3	C	$E3	c
$A4	$	$C4	D	$E4	d
$A5	%	$C5	E	$E5	e
$A6	&	$C6	F	$E6	f
$A7	’	$C7	G	$E7	g
$A8	(	$C8	H	$E8	h
$A9	)	$C9	I	$E9	i
$AA	*	$CA	J	$EA	j
$AB	+	$CB	K	$EB	k
$AC	,	$CC	L	$EC	l
$AD	–	$CD	M	$ED	m
$AE	•	$CE	N	$EE	n
$AF	/	$CF	O	$EF	o
$B0	0	$D0	P	$F0	p
$B1	1	$D1	Q	$F1	q
$B2	2	$D2	R	$F2	r
$B3	3	$D3	S	$F3	s
$B4	4	$D4	T	$F4	t
$B5	5	$D5	U	$F5	u
$B6	6	$D6	V	$F6	v
$B7	7	$D7	W	$F7	w
$B8	8	$D8	X	$F8	x

Table 12–2: Continued

ASCII Code	*Symbol*	*ASCII Code*	*Symbol*	*ASCII Code*	*Symbol*
$B9	9	$D9	Y	$F9	y
$BA	:	$DA	Z	$FA	z
$BB	;	$DB	[	$FB	{
$BC	<	$DC	\	$FC	|
$BD	=	$DD	]	$FD	}
$BE	>	$DE	^	$FE	~
$BF	?	$DF	–	$FF	[rubout]

the code for the new character. There are also commands for turning the cursor on and off: use $86 (CURSORON) to turn it on, and $85 (CURSOROFF) to turn it off.

If you are using the Pascal driver, you can position the cursor by sending a $9E code (GOTOXY) followed by bytes representing the horizontal and vertical positions. These positions are offset from their actual positions by 32 so that position (2,4) is represented by (34,36), for example

The BASIC screen driver does not support a cursor-positioning command. To serve this purpose, you will have to call a subroutine like the one in listing 12-3. On entry to the subroutine, the horizontal position must be in X and the vertical position in Y.

To understand how this subroutine works, consider that the GS BASIC 80-column video driver keeps track of the current cursor position in three locations in page zero and page five of bank $00:

- CH ($24) horizontal cursor position (0 to 79)
- OURCH ($57B) duplicate of horizontal cursor position
- CV ($25) vertical cursor position (0 to 23)

To change the cursor position, you must place the new horizontal position in both CH and OURCH and the new vertical position in CV. (The position is automatically updated as you send characters to the screen.) There is a complication that arises when CV is changed, however. The BASIC driver continues to use the old CV value until the cursor actually moves to another line. To force an immediate repositioning,

Table 12–3: The 80-Column Text Screen Parameters for Windowing and Cursor Control

Bank $00 Address	*Symbolic Name*	*Description*
$20	WNDLFT	Left column of window (0 to 79)
$21	WNDWDTH	Width of window (1 to 80)
$22	WNDTOP	Top row of window (0 to 23)
$23	WNDBTM	Bottom row of window plus one (1 to 24)
$24	CH	Horizontal cursor position (0 to 79)
$25	CV	Vertical cursor position (0 to 23)
$57B	OURCH	Horizontal cursor position (0 to 79)

NOTE: WNDWDTH must not exceed 80 minus the value of WNDLFT.

you must place the desired value minus 1 in CV and then send a carriage return code to the output device. This is exactly what the GOTOXY subroutine does.

The GOTOXY subroutine also shows how to force long addressing (by putting a > in front of the label for an address) so that you do not have to change the direct page before writing to CH or CV. It also temporarily switches to an 8-bit accumulator so that when you store the accumulator, the next location in memory is not overwritten.

Window Dimensions

The 80-column BASIC video driver is normally able to display characters anywhere on the screen. You can, however, tell it to confine its operations to any rectangular window within the screen by redefining four window boundary parameters located in page zero of bank $00. These parameters are described in table 12-3.

The usual values for the parameters in table 12–3 are:

WNDLFT	0 (left edge is column 0)
WNDWDTH	80 (screen width is 80 columns)
WNDTOP	0 (top line is line 0)
WNDBTM	24 (bottom line + 1 is line 24)

As when changing CH or CV, you must switch to an 8-bit accumulator before storing a number to any of the window parameter locations.

One parameter you must handle carefully is WNDWDTH. It must not exceed the difference between 80 (the maximum window width) and WNDLFT (the left edge). Keep this in mind whenever you change WNDLFT.

Note that the window dimensioning techniques do not work when the Pascal video driver is being used. They only work for the BASIC driver.

PRINTER OUTPUT

Printer drivers also support a variety of commands, mostly for setting formatting parameters that affect the appearance of printer output but also for setting up the communications link with the printer. The major commands supported by the drivers for the built-in serial ports on the GS are summarized in table 12–4. (Driver commands more appropriate to modem interaction are not given in this table.) To use a command, you begin by sending a Control-I code (ASCII $09) to the driver. You can change this attention code to another control character with a Control-I Control-x sequence, where "Control-x" represents the new attention code.

Default settings for most of the printer parameters can be set with the Printer Port command in the Control Panel desk accessory.

The drivers for the GS's internal serial ports (which can also be used with modems and other serial devices) are not as flexible as some found in many non-Apple printer interfaces. Such interfaces often support several commands that let you print the contents of the text or graphics screens in many different formats and sizes with a few simple commands. Without these commands, an application programmer must write his own routines.

Keep in mind that the printer itself may support commands that the printer interface does not. For example, to enable special printing modes, such as boldfacing or underlining, normally you send commands directly to the printer (through the printer driver, of course). If the command conflicts with a driver command, you may have to first send a command to the driver that tells it to ignore control character interpretation; for the GS serial port, this is the Z command. If you do this, you can restore control character interpretation by calling the InitTextDev function.

THE TEXT TOOL SET IN ACTION

The program in listing 12–4 shows how to implement many of the techniques described in this chapter. In particular, it shows how to use the PRINT and PRINTLN macros, how to enable and display MouseText characters, how to position the screen cursor, and how to turn on the printer.

Table 12–4: Control Commands Supported by the Driver for the GS Serial Ports. For a Printer Port, Commands Must Be Preceded by Character $89 (Control-I).

Command String	*Action*

nB	Set the serial baud rate.
	n=0 default n=8 1200 baud
	n=1 50 baud n=9 1800 baud
	n=2 75 baud n=10 2400 baud
	n=3 110 baud n=11 3600 baud
	n=4 135 baud n=12 4800 baud
	n=5 150 baud n=13 7200 baud
	n=6 300 baud n=14 9600 baud
	n=7 600 baud n=15 19200 baud

nD	Set the number of data bits and stop bits.
	n=0 8 data + 1 stop n=4 8 data + 2 stop
	n=1 7 data + 1 stop n=5 7 data + 2 stop
	n=2 6 data + 1 stop n=6 6 data + 2 stop
	n=3 5 data + 1 stop n=7 5 data + 2 stop

nP	Set the parity.
	n=0 no parity bit n=2 no parity bit
	n=1 odd parity n=3 even parity

nN	Set the line length to n.
CE CD	Enable (CE) or disable (CD) line formatting. When enabled, the GS inserts a carriage return at the end of a line (the length is set by the nN command).
AE AD	Enable (AE) or disable (AD) proper operation of the Applesoft BASIC TAB commands. When enabled, the line length is forced to the video screen width.

Table 12–4: Continued

Command String	*Action*
BE BD	Enable (BE) or disable (BD) the buffering of incoming and outgoing characters. The buffers are each 2048 bytes in size. Characters are sent from the output buffer in response to an interrupt indicating that the peripheral device is ready to receive a character.
XE XD	Enable (XE) or disable (XD) the XON/XOFF software handshake protocol. When enabled, and the firmware receives an XOFF character (ASCII $93), characters are not transmitted until an XON character (ASCII $91) is received. Some printers use this technique for flow control.
LE LD	Enable (LE) or disable (LD) the sending of a line feed character (ASCII $8A) after a carriage return character (ASCII $8D). Enable this feature if your printer seems to print everything on one line; disable it if the printer double-spaces all output.
R	Reset the serial port. This restores the default parameter settings set in the Control Panel.
Z	Zap control character interpretation. After this command, the firmware ignores character sequences which would normally be interpreted as commands.

REFERENCE SECTION

Table R12–1: The Major Functions in the Text Tool Set ($0C)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
ErrWriteBlock	$1F	TextPtr (L)	Ptr to block of text
		Offset (W)	Offset to first character
		Count (W)	Number of characters

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
ErrWriteChar	$19	Char (W)	ASCII character to write
ErrWriteCString	$21	CStringPtr (L)	Ptr to C-style text string
ErrWriteLine	$1B	StringPtr (L)	Ptr to text string
ErrWriteString	$1D	StringPtr (L)	Ptr to text string
GetErrorDevice	$14	result (W)	Driver type code
		result (L)	Output slot (or vector)
GetErrGlobals	$0E	result (W)	Current error AND mask
		result (W)	Current error OR mask
GetInGlobals	$0C	result (W)	Current input AND mask
		result (W)	Current input OR mask
GetInputDevice	$12	result (W)	Device type code
		result (L)	Output slot (or vector)
GetOutGlobals	$0D	result (W)	Current output AND mask
		result (W)	Current output OR mask
GetOutputDevice	$13	result (W)	Driver type code
		result (L)	Output slot (or vector)
InitTextDev	$15	DeviceNum (W)	Device number
ReadChar	$22	result (W)	Inputted character code
		EchoFlag (W)	1 = echo/0 = don't echo

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
ReadLine	$24	result (W)	Character count
		BufferPtr (L)	Ptr to input buffer
		MaxCount (W)	Size of buffer
		EOLChar (W)	End-of-line character
		EchoFlag (W)	1 = echo/0 = don't echo
SetErrorDevice	$11	DeviceType (W)	Driver type code
		SlotOrAddr (L)	Error output slot (or address)
SetErrGlobals	$0B	ANDMask (W)	Error output AND mask
		ORMask (W)	Error output OR mask
SetInGlobals	$09	ANDMask (W)	Input AND mask
		ORMask (W)	Input OR mask
SetInputDevice	$0F	DeviceType (W)	Device type code
		SlotOrAddr (L)	Input slot (or address)
SetOutGlobals	$0A	ANDMask (W)	Output AND mask
		ORMask (W)	Output OR mask
SetOutputDevice	$10	DeviceType (W)	Device type code
		SlotOrAddr (L)	Output slot (or address)
TextReadBlock	$23	BlockPtr (L)	Ptr to start of text block
		Offset (W)	Offset to starting position
		Count (W)	Number of characters to read
		EchoFlag (W)	1 = echo/0 = don't echo

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
TextShutDown	$03	[no parameters]	
TextStartup	$02	[no parameters]	
TextWriteBlock	$1E	TextPtr (L)	Ptr to block of text
		Offset (W)	Offset to first character
		Count (W)	Number of characters
WriteChar	$18	Char (W)	ASCII character to write
WriteCString	$20	CStringPtr (L)	Ptr to C-style text string
WriteLine	$1A	StringPtr (L)	Ptr to text string
WriteString	$1C	StringPtr (L)	Ptr to text string

Table R12–2: The Text Tool Set Error Codes

Error Code	*Description of Error Condition*
$0C01	Illegal device type code.
$0C02	Illegal device number.
$0C03	Illegal operation.
$0C04	Undefined hardware error.
$0C05	The device is no longer on-line.

Table R12–3: Useful Functions in the Miscellaneous Tool Set ($03)

Function Name	*Function Number*	*Stack Parameters*	*Description of Parameter*
GetIRQEnable	$29	result (W)	Interrupt status word
IntSource	$23	IntControl (W)	Interrupt control word

Listing 12–1: Character Device Initialization Subroutines

```
; Enable the 80-column video display for output and the
; 80-column keyboard input subroutine.

VIDEO_ON    START

            PEA      0                   ;BASIC device driver
            PushLong #3                  ;Select slot #3 for output
            _SetOutputDevice

            PEA      1                   ;Initialize output device
            _InitTextDev

            PEA      0                   ;BASIC-type device
            PushLong #3                  ;Select slot #3 for input
            _SetInputDevice

            PEA      0                   ;Initialize input device
            _InitTextDev

            PEA      $FF                 ;AND mask: do nothing
            PEA      $80                 ;OR mask: set high bit
            _SetOutGlobals

            PEA      $FF                 ;AND mask: do nothing
            PEA      $80                 ;OR mask: set high bit
            _SetInGlobals

            RTS

            END

; Enable the printer in slot 1. This is equivalent
; to a PR#1 in the good old days.

PRINT_ON    START

            PEA      0                   ;BASIC device driver
            PushLong #1                  ;Select slot #1 for output
            _SetOutputDevice

            PEA      1                   ;Initialize the output device
            _InitTextDev

            RTS

            END
```

Listing 12–2: The PRINT and PRINTLN Macros for Sending Text to the Active Output Device

```
; PRINT 'text phrase'
;
; This prints a line of text without a trailing CR.

         MACRO
&lab     PRINT &text

&lab     PEA   t&syscnt|-16      ;Push pointer (high)
         PEA   t&syscnt          ;Push pointer (low)
         LDX   #$1C0C            ;WriteString function
         JSL   $E10000           ;Tool entry point
         BRA   e&syscnt

t&syscnt DC    I1'L:&text'       ;Length of text
         DC    C"&text"          ;The text itself

e&syscnt ANOP
         MEND


; PRINTLN 'text phrase'
;
; This prints a line of text followed by a CR.

         MACRO
&lab     PRINTLN &text

&lab     PEA   t&syscnt|-16      ;Push pointer (high)
         PEA   t&syscnt          ;Push pointer (low)
         LDX   #$1A0C            ;WriteLine function
         JSL   $E10000           ;Tool entry point
         BRA   e&syscnt

t&syscnt DC    I1'L:&text'       ;Length of text
         DC    C"&text"          ;The text itself

e&syscnt ANOP
         MEND
```

Listing 12–3: The GOTOXY Subroutine for the 80-column Text Screen

```
; GOTOXY Subroutine. Enter with the horizontal position in X
; and the vertical position in Y.

CH       GEQU    $24             ;Cursor horizontal
CV       GEQU    $25             ;Cursor vertical
OURCH    GEQU    $57B            ;Cursor horizontal
```

```
CR         GEQU    $8D                ;Carriage return code

GOTOXY     START

           PHX                        ;Save horizontal position

           DEY                        ;Reduce vertical coordinate by 1
           TYA                        ; and put it in the A register.

           SEP     #$20               ;8-bit accumulator
           LONGA   OFF

           STA     >CV                ;Monitor's CV (cursor vertical)

           REP     #$20               ;Back to 16-bit accumulator
           LONGA   ON

           PEA     CR                 ;(do this first to make sure
           _WriteChar                 ; the vertical change takes hold)

           PLA                        ;Get horizontal position

           SEP     #$20               ;8-bit accumulator
           LONGA   OFF

           STA     >CH                ;Monitor's CH (cursor horizontal)
           STA     >OURCH             ;CH for 80-column firmware

           REP     #$20               ;Back to 16-bit accumulator
           LONGA   ON

           RTS

           END
```

Listing 12–4: A Program for Exercising the Text Tools

```
*****************************
* Exploring the Text Tools *
*****************************
           LIST    OFF
           SYMBOL  OFF
           ABSADDR ON
           INSTIME ON
           GEN     ON

           KEEP    TEXT               ;Object code file
           MCOPY   TEXT.MAC           ;Macro file

MyCode     START
```

```
;Monitor zero page locations:

WNDLFT     GEQU    $20              ;Left edge of window
WNDWDTH    GEQU    $21              ;Width: must be <= 80-WNDLFT
WNDTOP     GEQU    $22              ;Top line
WNDBTM     GEQU    $23              ;Bottom line + 1
CH         GEQU    $24              ;Cursor horizontal
CV         GEQU    $25              ;Cursor vertical
OURCH      GEQU    $57B             ;Cursor horizontal

;Special video commands:

CURSOROFF  GEQU    $85              ;Cursor off
CURSORON   GEQU    $86              ;Cursor on
BEEP       GEQU    $87              ;Beep the speaker
BACKSPACE  GEQU    $88              ;Move cursor left
LF         GEQU    $8A              ;Move cursor down
CLREOP     GEQU    $8B              ;Clear to end of screen
CLRSCREEN  GEQU    $8C              ;Clear the entire screen
CR         GEQU    $8D              ;Carriage return
NORMAL     GEQU    $8E              ;Normal video
INVERSE    GEQU    $8F              ;Inverse video
SET_40COL  GEQU    $91              ;Switch to 40-column display
SET_80COL  GEQU    $92              ;Switch to 80-column display
PAUSE      GEQU    $93              ;Stop output until key pressed
FIRM_OFF   GEQU    $95              ;Turn off enhanced video firmware
SCROLLD    GEQU    $96              ;Scroll down
SCROLLU    GEQU    $97              ;Scroll up
MTXT_OFF   GEQU    $98              ;MouseText off
HOME       GEQU    $99              ;Move cursor to top left corner
CLRLINE    GEQU    $9A              ;Clear line
MTXT_ON    GEQU    $9B              ;MouseText on
MOVERIGHT  GEQU    $9C              ;Move cursor right
CLREOL     GEQU    $9D              ;Clear to end of line
SET_CURS   GEQU    $9E              ;Set cursor (BASIC only)
MOVEUP     GEQU    $9F              ;Move cursor up

           PHK                      ;Set data bank = program bank so we
           PLB                      ; can use absolute addressing.

           _TLStartup               ;Tool locator
           _MTStartup               ;Miscellaneous Tools

           PHA
           _MMStartup               ;Memory Manager
           PLA
           STA     MyID

           _TextStartup             ;Text Tools

           JSR     VIDEO_ON         ;Set up 80-column I/O
```

```
; Just for fun, let's display the entire
; MouseText character set:

            PRINTLN 'Here is the MouseText character set:'
            PEA     CR
            _WriteChar

            PEA     MTXT_ON
            _WriteChar                ;Enable MouseText
            PEA     INVERSE
            _WriteChar                ;Select inverse mode

            LDA     #$C0              ;First MouseText character
ShowIcon    PHA
            PHA
            _WriteChar                ;Display it
            PLA
            INC     A
            CMP     #$E0              ;At the end?
            BNE     ShowIcon          ;No, so branch

            PEA     NORMAL
            _WriteChar                ;Select normal mode
            PEA     MTXT_OFF
            _WriteChar                ;Disable MouseText

            PEA     CR
            _WriteChar
            PEA     CR
            _WriteChar

; GOTOXY positions the cursor:

            LDX     #10               ;Horizontal position
            LDY     #8                ;Vertical position
            JSR     GOTOXY
            PRINTLN '<- - - coordinate (10,8)'

Get_Mode    PEA     CR
            _WriteChar
            PEA     BEEP
            _WriteChar
            PRINT   'Send output to [S]creen or [P]rinter? '

            PHA
            PEA      1                ;echo the input
            _ReadChar                 ;Read keyboard input
            PLA
```

```
          CMP    #'S'
          BEQ    ShowMsg
          CMP    #'s'
          BEQ    ShowMsg

          CMP    #'P'
          BEQ    PrintMsg
          CMP    #'p'
          BNE    Get_Mode

PrintMsg  JSR    PRINT_ON             ;Enable the printer

ShowMsg   PEA    CR
          _WriteChar
          PEA    CR
          _WriteChar

          PRINTLN 'PRINTLN is a macro which displays a line'
          PRINTLN 'of text followed by a Carriage Return.'

          PRINT  'PRINT does the same thing, except there'
          PEA    CR
          _WriteChar
          PRINT  'is no trailing Carriage Return.'
          PEA    CR
          _WriteChar

          PEA    CR
          _WriteChar

          PRINT  'Press any key to Quit: '
          PHA
          PEA     $0000               ;No echo
          _ReadChar                   ;Wait for keypress
          PLA

          JSR     VIDEO_ON            ;Back to 80-column screen

;Shut down the tool sets we used:

          _TextShutDown

          PushWord MyID
          _MMShutDown

          _MTShutDown
          _TLShutDown

          _QUIT QuitParms

          BRK    $F0                  ;(should never get here)
```

```
QuitParms  DC       I4'0'
           DC       I2'$0000'          ;Not restartable

MyID       DS       2

           END


; Use the COPY directive to include the PRINT_ON and
; VIDEO_ON subroutines found in listing 12-1 and
; the GOTOXY subroutine in listing 12-3.
```

APPENDIX 1

ASCII Character Codes

Characters such as letters and digits are stored inside the GS in binary form (a series of 1s and 0s), the only form a computer understands. The GS system software uses the ASCII (American Standard Code for Information Interchange) scheme for assigning characters and symbols to numeric codes. In fact, almost all microcomputer software uses the ASCII standard.

In the ASCII standard, each character is represented by a series of seven 1s and 0s. These binary digits occupy the first seven bits of the byte in which a character is stored. This means that 128 unique codes are available, enough to account for all the printable symbols on a standard typewriter keyboard and a few more. The unused bit in a byte is usually set to 0, but if you are sending characters to the text screen of the GS, it must be set to 1 (see chapter 12).

The system font, which QuickDraw II uses when it draws text on the super high-resolution graphics screen, assigns special symbols to the 128 codes which have the high bit set to 1. These symbols are not defined by the ASCII standard and can represent any symbol the designer of the font wishes.

The first 32 ASCII characters are called *control characters*. They are not supposed to represent visible symbols; instead, they are to be sent to a video display device or a printer controller to cause it to perform some special action such as ringing a bell (ASCII code 7, bell), moving the cursor or print head to the beginning of a line (ASCII code 13, carriage return), or moving the cursor or print head down one line (ASCII code 10, linefeed).

Although the driver for the GS 80-column text screen reacts to most control characters in standard ways (see chapter 12), the QuickDraw II text-drawing functions do not (see chapter 6). Instead, QuickDraw attempts to draw a symbol that was assigned to the code when the font was defined. For the system font, four control characters are associated with special symbols:

$11	Open-Apple icon
$12	Checkmark icon
$13	Diamond icon
$14	Solid-Apple icon

Other control characters in the system font are blanks or are undefined. (Undefined characters appear as white question marks on a black background.)

The following table shows the standard symbols for all 128 ASCII character codes. It also indicates how you can enter each character code from the keyboard.

Note that in certain situations, the GS displays MouseText icons on the text screen instead of the standard symbols for codes $40 to $5F. See chapter 12 for an explanation of when it does this.

Table of ASCII Character Codes

ASCII code			
Dec	*Hex*	*Symbol (or mnemonic)*	*Keys to press*
000	$00	NUL (Null)	Control @
001	$01	SOH (Start of header)	Control A
002	$02	STX (Start of text)	Control B
003	$03	ETX (End of text)	Control C
004	$04	EOT (End of transmission)	Control D
005	$05	ENQ (Enquiry)	Control E
006	$06	ACK (Acknowledge)	Control F
007	$07	BEL (Bell)	Control G
008	$08	BS (Backspace)	Control H or Left-arrow
009	$09	HT (Horizontal tabulation)	Control I or Tab
010	$0A	LF (Line feed)	Control J or Down-arrow
011	$0B	VT (Vertical tabulation)	Control K or Up-arrow
012	$0C	FF (Form feed)	Control L

ASCII code			
Dec	*Hex*	*Symbol (or mnemonic)*	*Keys to press*
013	$0D	CR (Carriage return)	Control M or Return
014	$0E	SO (Shift out)	Control N
015	$0F	SI (Shift in)	Control O
016	$10	DLE (Data link escape)	Control P
017	$11	DC1 (Device control 1)	Control Q
018	$12	DC2 (Device control 2)	Control R
019	$13	DC3 (Device control 3)	Control S
020	$14	DC4 (Device control 4)	Control T
021	$15	NAK (Negative acknowledge)	Control U or Right-arrow
022	$16	SYN (Synchronous idle)	Control V
023	$17	ETB (End transmission block)	Control W
024	$18	CAN (Cancel)	Control X
025	$19	EM (End of medium)	Control Y
026	$1A	SUB (Substitute)	Control Z
027	$1B	ESC (Escape)	Control [or Esc
028	$1C	FS (Field separator)	Control \
029	$1D	GS (Group separator)	Control]
030	$1E	RS (Record separator)	Control ˆ
031	$1F	US (Unit separator)	Control _
032	$20	(space)	Space Bar
033	$21	!	Shift 1
034	$22	”	Shift ’
035	$23	#	Shift 3
036	$24	$	Shift 4

ASCII code			
Dec	*Hex*	*Symbol (or mnemonic)*	*Keys to press*
037	$25	%	Shift 5
038	$26	&	Shift 7
039	$27	’	’
040	$28	(	Shift 9
041	$29	)	Shift 0
042	$2A	*	Shift 8
043	$2B	+	Shift =
044	$2C	,	,
045	$2D		
046	$2E	.	.
047	$2F	/	/
048	$30	0	0
049	$31	1	1
050	$32	2	2
051	$33	3	3
052	$34	4	4
053	$35	5	5
054	$36	6	6
055	$37	7	7
056	$38	8	8
057	$39	9	9
058	$3A	:	Shift ;
059	$3B	;	;
060	$3C	<	Shift ,
061	$3D	=	=
062	$3E	>	Shift .

ASCII code			
Dec	*Hex*	*Symbol (or mnemonic)*	*Keys to press*
063	$3F	?	Shift /
064	$40	@	Shift 2
065	$41	A	Shift A
066	$42	B	Shift B
067	$43	C	Shift C
068	$44	D	Shift D
069	$45	E	Shift E
070	$46	F	Shift F
071	$47	G	Shift G
072	$48	H	Shift H
073	$49	I	Shift I
074	$4A	J	Shift J
075	$4B	K	Shift K
076	$4C	L	Shift L
077	$4D	M	Shift M
078	$4E	N	Shift N
079	$4F	O	Shift O
080	$50	P	Shift P
081	$51	Q	Shift Q
082	$52	R	Shift R
083	$53	S	Shift S
084	$54	T	Shift T
085	$55	U	Shift U
086	$56	V	Shift V
087	$57	W	Shift W
088	$58	X	Shift X

ASCII code			
Dec	*Hex*	*Symbol (or mnemonic)*	*Keys to press*
089	$59	Y	Shift Y
090	$5A	Z	Shift Z
091	$5B	[	[
092	$5C	\	\
093	$5D	]	]
094	$5E	^	Shift 6
095	$5F	_	Shift -
096	$60	‘	‘
097	$61	a	A
098	$62	b	B
099	$63	c	C
100	$64	d	D
101	$65	e	E
102	$66	f	F
103	$67	g	G
104	$68	h	H
105	$69	i	I
106	$6A	j	J
107	$6B	k	K
108	$6C	l	L
109	$6D	m	M
110	$6E	n	N
111	$6F	o	O
112	$70	p	P
113	$71	q	Q
114	$72	r	R

ASCII code			
Dec	*Hex*	*Symbol (or mnemonic)*	*Keys to press*
115	$73	s	S
116	$74	t	T
117	$75	u	U
118	$76	v	V
119	$77	w	W
120	$78	x	X
121	$79	y	Y
122	$7A	z	Z
123	$7B	{	Shift [
124	$7C	|	Shift \
125	$7D	}	Shift]
126	$7E	~	Shift ‘
127	$7F	(rubout)	Delete

APPENDIX 2

The Western Design Center W65C816 Data Sheet

CMOS W65C816 and W65C802 16-Bit Microprocessor Family

Features

- Advanced CMOS design for low power consumption and increased noise immunity
- Single 3-6V power supply, 5V specified
- Emulation mode allows complete hardware and software compatibility with 6502 designs
- 24-bit address bus allows access to 16 MBytes of memory space
- Full 16-bit ALU, Accumulator, Stack Pointer, and Index Registers
- Valid Data Address (VDA) and Valid Program Address (VPA) output allows dual cache and cycle steal DMA implementation
- Vector Pull (VP) output indicates when interrupt vectors are being addressed. May be used to implement vectored interrupt design
- Abort (ABORT) input and associated vector supports virtual memory system design
- Separate program and data bank registers allow program segmentation or full 16-MByte linear addressing
- New Direct Register and stack relative addressing provides capability for re-entrant, re-cursive and re-locatable programming
- 24 addressing modes—13 original 6502 modes, plus 11 new addressing modes with 91 instructions using 255 opcodes
- New Wait for Interrupt (WAI) and Stop the Clock (STP) instructions further reduce power consumption, decrease interrupt latency and allows synchronization with external events
- New Co-Processor instruction (COP) with associated vector supports co-processor configurations, i.e., floating point processors
- New block move ability

General Description

WDC's W65C802 and W65C816 are CMOS 16-bit microprocessors featuring total software compatibility with their 8-bit NMOS and CMOS 6500-series predecessors. The W65C802 is pin-to-pin compatible with 8-bit devices currently available, while the W65C816 extends addressing to a full 16 megabytes. These devices offer the many advantages of CMOS technology, including increased noise immunity, higher reliability, and greatly reduced power requirements. A software switch determines whether the processor is in the 8-bit "emulation" mode, or in the native mode, thus allowing existing systems to use the expanded features.

As shown in the processor programming model, the Accumulator, ALU, X and Y Index registers, and Stack Pointer register have all been extended to 16 bits. A new 16-bit Direct Page register augments the Direct Page addressing mode (formerly Zero Page addressing) Separate Program Bank and Data Bank registers allow 24-bit memory addressing with segmented or linear addressing.

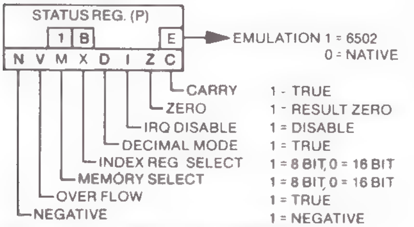
Four new signals provide the system designer with many options. The ABORT input can interrupt the currently executing instruction without modifying internal register, thus allowing virtual memory system design Valid Data Address (VDA) and Valid Program Address (VPA) outputs facilitate dual cache memory by indicating whether a data segment or program segment is accessed. Modifying a vector is made easy by monitoring the Vector Pull (VP) output.

Note: To assist the design engineer, a Caveat and Application Information section has been included within this data sheet

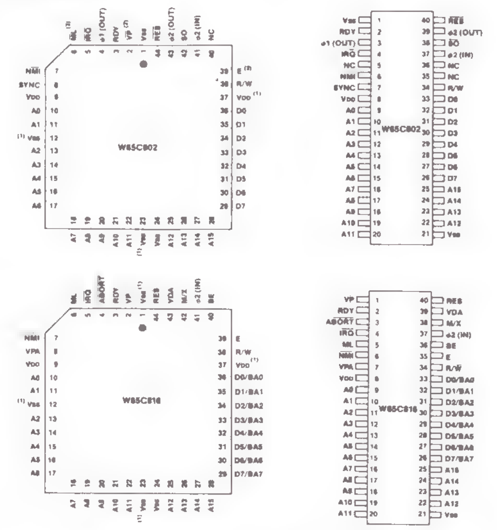
W65C816 Processor Programming Model

8 BITS	8 BITS		8 BITS
Data Bank Reg (DBR)	X Register Hi (XH)	(X)	X Register Low (XL)
Data Bank Reg (DBR)	Y Register Hi (YH)	(Y)	Y Register Low (YL)
00	Stack Register Hi (SH)	(S)	Stack Reg. Low (SL)
□ - 6502 Registers	Accumulator (B)	(C)	Accumulator (A)
Program Bank Reg (PBR)	Program (PCH)	(PC)	Counter (PCL)
00	Direct Reg Hi (DH)	(D)	Direct Reg. Low (DL)

Status Register Coding



Pin Configuration



W65C802
W65C802
W65C816
W65C816
Vss RDY φ1 (OUT) IRQ NC NMI SYNC VDD A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11
Vss A12 A13 A14 A15 D7 D6 D5 D4 D3 D2 D1 D0 R/W NC NC φ2 (IN) SO φ2 (OUT) RES
VP RDY ABORT IRQ ML NMI VPA VDD A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11
Vss A12 A13 A14 A15 D7/BA7 D6/BA6 D5/BA5 D4/BA4 D3/BA3 D2/BA2 D1/BA1 D0/BA0 R/W E BE φ2 (IN) M/X VDA RES

For notes, refer to Packaging Information section.

Design Engineer William D Mensch Jr

WDC THE WESTERN DESIGN CENTER, INC.
2166 East Brown Road • Mesa Arizona 85203 • 602 962 4545

Advance Information Data Sheet:
This is advanced information and specifications are subject to change without notice.

Absolute Maximum Ratings: (Note 1)

Rating	Symbol	Value
Supply Voltage	V_{DD}	-0 3V to +7 0V
Input Voltage	V_{IN}	-0 3V to V_{DD} +0 3V
Operating Temperature	T_A	0° C to +70° C
Storage Temperature	T_S	-55° C to +150° C

This device contains input protection against damage due to high static voltages or electric fields; however, precautions should be taken to avoid application of voltages higher than the maximum rating.

Notes

1. Exceeding these ratings may cause permanent damage. Functional operation under these conditions is not implied.

DC Characteristics (All Devices): V_{DD} = 5 0V ±5%, V_{SS} = 0V, T_A 0° C to +70° C

Parameter	Symbol	Min	Max	Unit
Input High Voltage RES, RDY, IRQ, Data, $\overline{\text{SO}}$, BE, φ2 (IN), NMI, ABORT	V_{IH}	 2 0 0 7 V_{DD}	 V_{DD} + 0 3 V_{DD} + 0 3	 V V
Input Low Voltage RES, RDY, IRQ, Data, $\overline{\text{SO}}$, BE, φ2 (IN), $\overline{\text{NMI}}$, $\overline{\text{ABORT}}$	V_{IL}	 -0 3 -0 3	 0 8 0 2	 V V
Input Leakage Current (V_{IN} - 0 to V_{DD}) RES, NMI, IRQ, SO, BE, ABORT (Internal Pullup) RDY (Internal Pullup, Open Drain) φ2 (IN) Address Data, R/$\overline{\text{W}}$ (Off State, BE 0)	I_{IN}	 100 100 -1 -10	 1 10 1 10	 μA μA μA μA
Output High Voltage (I_{OH} - 100μA) SYNC, Data, Address, R/W, ML, VP, M/X, E, VDA, VPA φ1 (OUT), φ2 (OUT)	V_{OH}	 0 7 V_{DD}		 V
Output Low Voltage (I_{OL} = 1 6mA) SYNC, Data, Address, R/W, ML, VP, M/X, E, VDA, VPA, φ1 (OUT), φ2 (OUT)	V_{OL}		 0 4	 V
Supply Current (No Load)	I_{DD}		4	mA/MHz
Standby Current (No Load, Data Bus =Vss or V_{DD} RES, NMI, IRQ, SO, BE, ABORT, φ2 = V_{DD})	I_{SB}	—	 10	 μA
Capacitance (V_{IN} 0V, T_A - 25° C f 2 MHz) Logic, φ2 (IN) Address, Data, R/$\overline{\text{W}}$ (Off State)	 C_{IN} C_{TS}	 —	 10 15	 pF pF

Pin Function Table

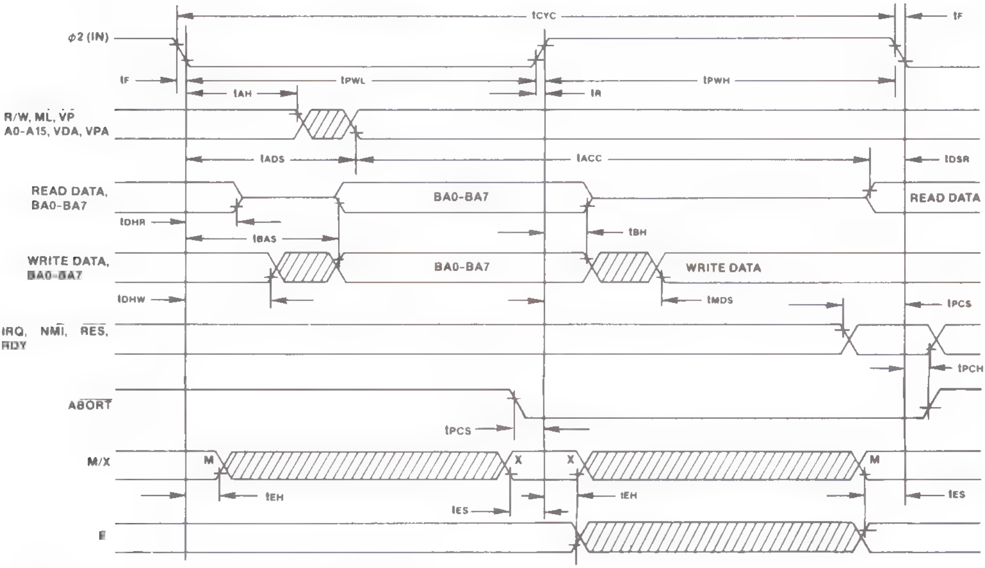
Pin	Description
A0-A15	Address Bus
$\overline{\text{ABORT}}$	Abort Input
BE	Bus Enable
φ2 (IN)	Phase 2 In Clock
φ1 (OUT)	Phase 1 Out Clock
φ2 (OUT)	Phase 2 Out Clock
D0-D7	Data Bus (G65SC802)
D0/BA0-D7/BA7	Data Bus, Multiplexed (G65SC816)
E	Emulation Select
$\overline{\text{IRQ}}$	Interrupt Request
$\overline{\text{ML}}$	Memory Lock
M/X	Mode Select (P_M or P_X)

Pin	Description
NC	No Connection
$\overline{\text{NMI}}$	Non-Maskable Interrupt
RDY	Ready
$\overline{\text{RES}}$	Reset
R/$\overline{\text{W}}$	Read/Write
$\overline{\text{SO}}$	Set Overflow
SYNC	Synchronize
VDA	Valid Data Address
$\overline{\text{VP}}$	Vector Pull
VPA	Valid Program Address
V_{DD}	Positive Power Supply (+5 Volts)
V_{SS}	Internal Logic Ground

AC Characteristics (W65C816): V_{DD} = 5 0V ±5%, V_{SS} = 0V, T_A = 0° C to +70° C

		2 MHz		4 MHz		6 MHz		8 MHz		
Parameter	Symbol	Min	Max	Min	Max	Min	Max	Min	Max	Unit
Cycle Time	t_{CYC}	500	DC	250	DC	167	DC	125	DC	nS
Clock Pulse Width Low	t_{PWL}	0 240	10	0 120	10	0 080	10	0 060	10	µS
Clock Pulse Width High	t_{PWH}	240	∞	120	∞	80	∞	60	∞	nS
Fall Time, Rise Time	t_F, t_R	—	10	—	10	—	5	—	5	nS
A0-A15 Hold Time	t_{AH}	10	—	10	—	10	—	10	—	nS
A0-A15 Setup Time	t_{ADS}	—	100	—	75	—	60	—	40	nS
BA0 BA7 Hold Time	t_{BH}	10	—	10	—	10	—	10	—	nS
BA0-BA7 Setup Time	t_{BAS}	—	100	—	90	—	65	—	45	nS
Access Time	t_{ACC}	365	—	130	—	87	—	70	—	nS
Read Data Hold Time	t_{DHR}	10	—	10	—	10	—	10	—	nS
Read Data Setup Time	t_{DSR}	40	—	30	-	20	-	15	—	nS
Write Data Delay Time	t_{MDS}		100	—	70	—	60	-	40	nS
Write Data Hold Time	t_{DHW}	10	—	10	—	10	—	10	—	nS
Processor Control Setup Time	t_{PCS}	40	—	30	—	20	—	15	—	nS
Processor Control Hold Time	t_{PCH}	10	—	10		10	-	10	—	nS
E,MX Output Hold Time	t_{EH}	10	—	10	—	5	—	5	—	nS
E,MX Output Setup Time	t_{ES}	50	—	50	—	25	—	15	—	nS
Capacitive Load (Address, Data, and R/W)	C_{EXT}	-	100	—	100		35	—	35	pF
BE to High Impedance State	t_{BHZ}	-	30		30		30	-	30	nS
BE to Valid Data	t_{BVD}	-	30	-	30	—	30	—	30	nS

Timing Diagram (W65C816)



Timing Notes:

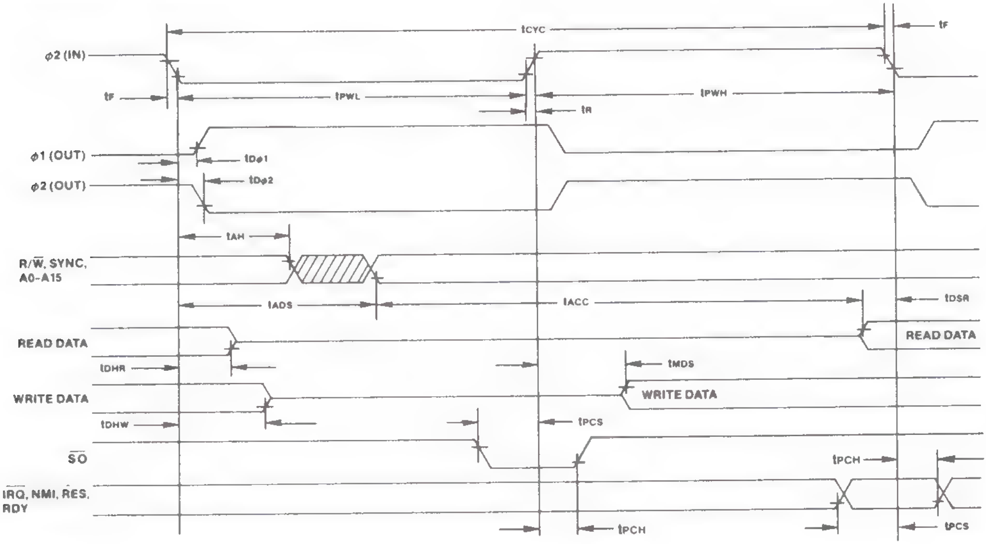
1 Voltage levels are $V_L < 0\ 4V$, $V_H > 2\ 4V$

2 Timing measurement points are 0 8V and 2 0V

AC Characteristics (W65C802): V_{DD} - 5 0V +5% V_{SS} 0V T_A 0° C to +70° C

		2 MHz		4 MHz		6 MHz		8 MHz		
Parameter	Symbol	Min	Max	Min	Max	Min	Max	Min	Max	Unit
Cycle Time	t_{CYC}	500	DC	250	DC	167	DC	125	DC	nS
Clock Pulse Width Low	t_{PWL}	0 240	10	0 120	10	0 080	10	0 060	10	μS
Clock Pulse Width High	t_{PWH}	240	∞	120	∞	80	∞	60	∞	nS
Fall Time, Rise Time	t_F t_R	—	10	—	10	—	5	—	5	nS
Delay Time, φ2 (IN) to φ1 (OUT)	$t_{D\phi 1}$	—	20		20	—	20	—	20	nS
Delay Time φ2 (IN) to φ2 (OUT)	$t_{D\phi 2}$	—	40	—	40	—	40	–	40	nS
Address Hold Time	t_{AH}	10		10	—	10		10	–	nS
Address Setup Time	t_{ADS}	—	100		75	—	60	—	40	nS
Access Time	t_{ACC}	365	–	130		87	–	70	—	nS
Read Data Hold Time	t_{DHR}	10	—	10	—	10	-	10	—	nS
Read Data Setup Time	t_{DSR}	40		30	—	20	—	15	—	nS
Write Data Delay Time	t_{MDS}	—	100		70	—	60	—	40	nS
Write Data Hold Time	t_{DHW}	10	-	10		10	-	10	—	nS
Processor Control Setup Time	t_{PCS}	40	—	30	–	20	—	15	–	nS
Processor Control Hold Time	t_{PCH}	10	—	10		10	--	10	—	nS
Capacitive Load (Address, Data, and R/W)	C_{EXT}		100	-	100		35	—	35	pF

Timing Diagram (W65C802)



Timing Notes:

1. Voltage levels are V_L < 0 4V, V_H > 2 4V
2. Timing measurement points are 0.8V and 2.0V

Functional Description

The W65C802 offers the design engineer the opportunity to utilize both existing software programs and hardware configurations, while also achieving the added advantages of increased register lengths and faster execution times. The W65C802's "ease of use" design and implementation features provide the designer with increased flexibility and reduced implementation costs In the Emulation mode, the W65C802 not only offers software compatibility, but is also hardware (pin-to-pin) compatible with 6502 designs... plus it provides the advantages of 16-bit internal operation in 6502-compatible applications The W65C802 is an excellent direct replacement microprocessor for 6502 designs

The W65C816 provides the design engineer with upward mobility and software compatibility in applications where a 16-bit system configuration is desired The W65C816's 16-bit hardware configuration, coupled with current software allows a wide selection of system applications In the Emulation mode, the W65C816 offers many advantages, including full software compatibility with 6502 coding In addition, the W65C816's powerful instruction set and addressing modes make it an excellent choice for new 16-bit designs

Internal organization of the W65C802 and W65C816 can be divided into two parts 1) The Register Section, and 2) The Control Section Instructions (or opcodes) obtained from program memory are executed by implementing a series of data transfers within the Register Section. Signals that cause data transfers to be executed are generated within the Control Section. Both the W65C802 and the W65C816 have a 16-bit internal architecture with an 8-bit external data bus.

Instruction Register and Decode

An opcode enters the processor on the Data Bus, and is latched into the Instruction Register during the instruction fetch cycle This instruction is then decoded, along with timing and interrupt signals, to generate the various Instruction Register control signals

Timing Control Unit (TCU)

The Timing Control Unit keeps track of each instruction cycle as it is executed. The TCU is set to zero each time an instruction fetch is executed, and is advanced at the beginning of each cycle for as many cycles as is required to complete the instruction Each data transfer between registers depends upon decoding the contents of both the Instruction Register and the Timing Control Unit.

Arithmetic and Logic Unit (ALU)

All arithmetic and logic operations take place within the 16-bit ALU In addition to data operations, the ALU also calculates the effective address for relative and indexed addressing modes. The result of a data operation is stored in either memory or an internal register. Carry, Negative, Overflow and Zero flags may be updated following the ALU data operation

Internal Registers (Refer to Programming Model)

Accumulators (A, B, C)

The Accumulator is a general purpose register which stores one of the operands, or the result of most arithmetic and logical operations. In the Native mode (E=0), when the Accumulator Select Bit (M) equals zero, the Accumulator is established as 16 bits wide (A + B = C). When the Accumulator Select Bit (M) equals one, the Accumulator is 8 bits wide (A) In this case, the upper 8 bits (B) may be used for temporary storage in conjunction with the Exchange Accumulator (XBA) instruction

Data Bank Register (DBR)

During modes of operation, the 8-bit Data Bank Register holds the default bank address for memory transfers. The 24-bit address is composed of the 16-bit instruction effective address and the 8-bit Data Bank address. The register value is multiplexed with the data value and is present on the Data/Address lines during the first half of a data transfer memory cycle for the W65C816. The Data Bank Register is initialized to zero during Reset.

Direct (D)

The 16-bit Direct Register provides an address offset for all instructions using direct addressing. The effective bank zero address is formed by adding the 8-bit instruction operand address to the Direct Register. The Direct Register is initialized to zero during Reset.

Index (X and Y)

There are two Index Registers (X and Y) which may be used as general purpose registers or to provide an index value for calculation of the effective address When executing an instruction with indexed addressing, the microprocessor fetches the opcode and the base address, and then modifies the address by adding the Index Register contents to the address prior to performing the desired operation. Pre-indexing or post-indexing of indirect addresses may be selected. In the Native mode (E=0), both Index Registers are 16 bits wide (providing the Index Select Bit (X) equals zero) If the Index Select Bit (X) equals one, both registers will be 8 bits wide, and the high byte is forced to zero.

Processor Status (P)

The 8-bit Processor Status Register contains status flags and mode select bits The Carry (C), Negative (N), Overflow (V), and Zero (Z) status flags serve to report the status of most ALU operations. These status flags are tested by use of Conditional Branch instructions. The Decimal (D), IRQ Disable (I), Memory/Accumulator (M), and Index (X) bits are used as mode select flags. These flags are set by the program to change microprocessor operations

The Emulation (E) select and the Break (B) flags are accessible only through the Processor Status Register. The Emulation mode select flag is selected by the Exchange Carry and Emulation Bits (XCE) instruction Table 1, W65C802 and W65C816 Mode Comparison, illustrates the features of the Native (E=0) and Emulation (E=1) modes The M and X flags are always equal to one in the Emulation mode. When an interrupt occurs during the Emulation mode, the Break flag is written to stack memory as bit 4 of the Processor Status Register.

Program Bank Register (PBR)

The 8-bit Program Bank Register holds the bank address for all instruction fetches. The 24-bit address consists of the 16-bit instruction effective address and the 8-bit Program Bank address. The register value is multiplexed with the data value and presented on the Data/Address lines during the first half of a program memory read cycle. The Program Bank Register is initialized to zero during Reset. The PHK instruction pushes the PBR register onto the Stack.

Program Counter (PC)

The 16-bit Program Counter Register provides the addresses which are used to step the microprocessor through sequential program instructions. The register is incremented each time an instruction or operand is fetched from program memory

Stack Pointer (S)

The Stack Pointer is a 16-bit register which is used to indicate the next available location in the stack memory area. It serves as the effective address in stack addressing modes as well as subroutine and interrupt processing The Stack Pointer allows simple implementation of nested subroutines and multiple-level interrupts. During the Emulation mode, the Stack Pointer high-order byte (SH) is always equal to one. The bank address for all stack operations is Bank zero.

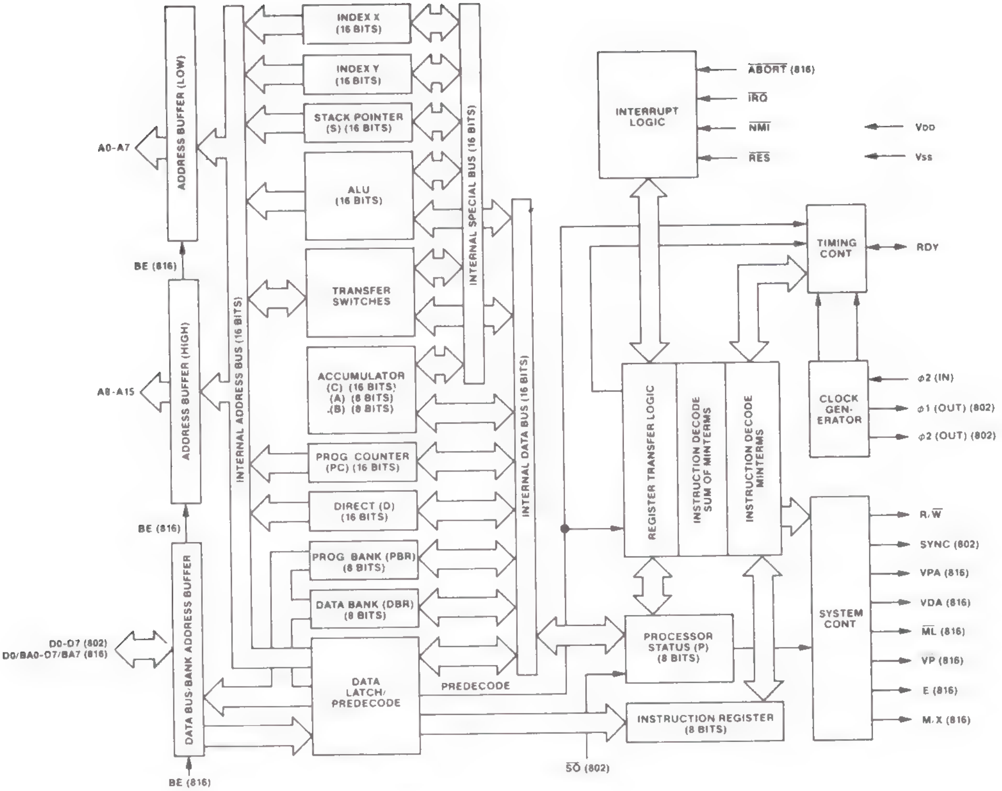


Figure 1. Block Diagram — Internal Architecture

Signal Description

The following Signal Description applies to both the W65C802 and the W65C816 except as otherwise noted

Abort (ABORT)—W65C816

The Abort input is used to abort instructions (usually due to an Address Bus condition). A negative transition will inhibit modification of any internal register during the current instruction. Upon completion of this instruction, an interrupt sequence is initiated. The location of the aborted opcode is stored as the return address in stack memory. The Abort vector address is 00FFF8,9 (Emulation mode) or 00FFE8,9 (Native mode). Note that ABORT is a pulse-sensitive signal; i.e., an abort will occur whenever there is a negative pulse (or level) on the ABORT pin during a φ2 clock.

Address Bus (A0-A15)

These sixteen output lines form the Address Bus for memory and I/O exchange on the Data Bus. When using the W65C816, the address lines may be set to the high impedance state by the Bus Enable (BE) signal.

Bus Enable (BE)—W65C816

The Bus Enable input signal allows external control of the Address and Data Buffers, as well as the R/$\overline{W}$ signal. With Bus Enable high, the R/$\overline{W}$ and Address Buffers are active. The Data/Address Buffers are active during the first half of every cycle and the second half of a write cycle. When BE is low, these buffers are disabled Bus Enable is an asynchronous signal

Data Bus (D0-D7)—W65C802

The eight Data Bus lines provide an 8-bit bidirectional Data Bus for use during data exchanges between the microprocessor and external memory or peripherals. Two memory cycles are required for the transfer of 16-bit values

Data/Address Bus (D0/BA0-D7/BA7)—W65C816

These eight lines multiplex address bits BA0-BA7 with the data value. The address is present during the first half of a memory cycle, and the data value is read or written during the second half of the memory cycle. Two memory cycles are required to transfer 16-bit values. These lines may be set to the high impedance state by the Bus Enable (BE) signal.

Emulation Status (E)—W65C816

The Emulation Status output reflects the state of the Emulation (E) mode flag in the Processor Status (P) Register. This signal may be thought of as an opcode extension and used for memory and system management.

Interrupt Request ($\overline{\text{IRQ}}$)

The Interrupt Request input signal is used to request that an interrupt sequence be initiated. When the IRQ Disable (I) flag is cleared, a low input logic level initiates an interrupt sequence after the current instruction is completed. The Wait for Interrupt (WAI) instruction may be executed to ensure the interrupt will be recognized immediately. The Interrupt Request vector address is 00FFFE,F (Emulation mode) or 00FFEE,F (Native mode). Since $\overline{\text{IRQ}}$ is a level-sensitive input, an interrupt will occur if the interrupt source was not cleared since the last interrupt. Also, no interrupt will occur if the interrupt source is cleared prior to interrupt recognition.

Memory Lock ($\overline{\text{ML}}$)—W65C816

The Memory Lock output may be used to ensure the integrity of Read-Modify-Write instructions in a multiprocessor system. Memory Lock indicates the need to defer arbitration of the next bus cycle. Memory Lock is low during the last three or five cycles of ASL, DEC, INC, LSR, ROL, ROR, TRB, and TSB memory referencing instructions, depending on the state of the M flag.

Memory/Index Select Status (M/X)—W65C816

This multiplexed output reflects the state of the Accumulator (M) and Index (X) select flags (bits 5 and 4 of the Processor Status (P) Register Flag M is valid during the Phase 2 clock negative transition and Flag X is valid during the Phase 2 clock positive transition. These bits may be thought of as opcode extensions and may be used for memory and system management

Non-Maskable Interrupt ($\overline{\text{NMI}}$)

A negative transition on the NMI input initiates an interrupt sequence. A high-to-low transition initiates an interrupt sequence after the current instruction is completed. The Wait for Interrupt (WAI) instruction may be executed to ensure that the interrupt will be recognized immediately The Non-Maskable Interrupt vector address is 00FFFA,B (Emulation mode) or 00FFEA B (Native mode) Since $\overline{\text{NMI}}$ is an edge-sensitive input, an interrupt will occur if there is a negative transition while servicing a previous interrupt. Also, no interrupt will occur if $\overline{\text{NMI}}$ remains low.

Phase 1 Out (ϕ1 (OUT))—W65C802

This inverted clock output signal provides timing for external read and write operations. Executing the Stop (STP) instruction holds this clock in the low state.

Phase 2 In (ϕ2 (IN))

This is the system clock input to the microprocessor internal clock generator (equivalent to ϕ0 (IN) on the 6502) During the low power Standby Mode, ϕ2 (IN) should be held in the high state to preserve the contents of internal registers

Phase 2 Out (ϕ2 (OUT))—W65C802

This clock output signal provides timing for external read and write operations. Addresses are valid (after the Address Setup Time (T_{ADS})) following the negative transition of Phase 2 Out Executing the Stop (STP) instruction holds Phase 2 Out in the High state

Read/Write (R/$\overline{\text{W}}$)

When the R/$\overline{\text{W}}$ output signal is in the high state, the microprocessor is reading data from memory or I/O. When in the low state, the Data Bus contains valid data from the microprocessor which is to be stored at the addressed memory location. When using the W65C816, the R/$\overline{\text{W}}$ signal may be set to the high impedance state by Bus Enable (BE).

Ready (RDY)

This bidirectional signal indicates that a Wait for Interrupt (WAI) instruction has been executed allowing the user to halt operation of the microprocessor. A low input logic level will halt the microprocessor in its current state (note that when in the Emulation mode, the W65C802 stops only during a read cycle). Returning RDY to the active high state allows the microprocessor to continue following the next Phase 2 In Clock negative transition. The RDY signal is internally pulled low following the execution of a Wait for Interrupt (WAI) instruction, and then returned to the high state when a $\overline{\text{RES}}$, $\overline{\text{ABORT}}$, $\overline{\text{NMI}}$, or $\overline{\text{IRQ}}$ external interrupt is provided. This feature may be used to eliminate interrupt latency by placing the WAI instruction at the beginning of the $\overline{\text{IRQ}}$ servicing routine. If the IRQ Disable flag has been set, the next instruction will be executed when the $\overline{\text{IRQ}}$ occurs. The processor will not stop after a WAI instruction if RDY has been forced to a high state. The Stop (STP) instruction has no effect on RDY

Reset ($\overline{\text{RES}}$)

The Reset input is used to initialize the microprocessor and start program execution. The Reset input buffer has hysteresis such that a simple R-C timing circuit may be used with the internal pullup device. The RES signal must be held low for at least two clock cycles after V_{DD} reaches operating voltage. Ready (RDY) has no effect while RES is being held low. During this Reset conditioning period, the following processor initialization takes place:

Registers

D	-	0000	SH	=	01
DBR	-	00	XH	=	00
PBR		00	YH	-	00

		N	V	M	X	D	I	Z	C/E	
P	-	*	*	1	1	0	1	*	*/1	* - Not Initialized

STP and WAI instructions are cleared.

Signals

E	-	1	VDA	=	0
M/X	-	1	$\overline{\text{VP}}$	-	1
R/$\overline{\text{W}}$	=	1	VPA	=	0
SYNC	-	0			

When $\overline{\text{Reset}}$ is brought high, an interrupt sequence is initiated:

- R/$\overline{\text{W}}$ remains in the high state during the stack address cycles.
- The Reset vector address is 00FFFC,D

Set Overflow ($\overline{\text{SO}}$)—W65C802

A negative transition on this input Sets the Overflow (V) flag, bit 6 of the Processor Status (P) Register.

Synchronize (SYNC)—W65C802

The SYNC output is provided to identify those cycles during which the microprocessor is fetching an opcode The SYNC signal is high during an opcode fetch cycle, and when combined with Ready (RDY), can be used for single instruction execution.

Valid Data Address (VDA) and Valid Program Address (VPA)—W65C816

These two output signals indicate valid memory addresses when high (logic 1), and must be used for memory or I/O address qualification.

VDA	VPA	
0	0	Internal Operation—Address and Data Bus available. The Address Bus may be invalid.
0	1	Valid program address—may be used for program cache control
1	0	Valid data address—may be used for data cache control.
1	1	Opcode fetch—may be used for program cache control and single step control.

V_{DD} and V_{SS}

V_{DD} is the positive supply voltage and V_{SS} is system logic ground. Pin 21 of the two V_{SS} pins on the W65C802 should be used for system ground.

Vector Pull ($\overline{\text{VP}}$)—W65C816

The Vector Pull output indicates that a vector location is being addressed during an interrupt sequence. $\overline{\text{VP}}$ is low during the last two interrupt sequence cycles, during which time the processor reads the interrupt vector. The $\overline{\text{VP}}$ signal may be used to select and prioritize interrupts from several sources by modifying the vector addresses.

Table 1. W65C816 Compatibility Issues

	W65C816/802	W65C02	NMOS 6502
1 S (Stack)	Always page 1 (E = 1), 8 bits 16 bits when (E = 0)	Always page 1, 8 bits	Always page 1, 8 bits
2 X (X Index Register)	Indexed page zero always in page 0 (E = 1), Cross page (E = 0)	Always page 0	Always page 0
3. Y (Y Index Register)	Indexed page zero always in page 0 (E = 1), Cross page (E = 0)	Always page 0	Always page 0
4 A (Accumulator)	8 bits (M = 1), 16 bits (M = 0)	8 bits	8 bits
5 P (Flag Registor)	N, V, and Z flags valid in decimal mode D = 0 after reset or interrupt	N, V, and Z flags valid in decimal mode D = 0 after reset and interrupt	N, V, and Z flags invalid in decimal mode D = unknown after reset D not modified after interrupt
6. Timing A. ABS, X ASL, LSR, ROL, ROR With No Page Crossing	7 cycles	6 cycles	7 cycles
B. Jump Indirect Operand = XXFF	5 cycles	6 cycles	5 cycles and invalid page crossing
C. Branch Across Page	4 cycles (E = 1) 3 cycles (E - 0)	4 cycles	4 cycles
D Decimal Mode	No additional cycle	Add 1 cycle	No additional cycle
7 BRK Vector	00FFFE,F (E 1) BRK bit 0 on stack if $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$, $\overline{\text{ABORT}}$ 00FFE6, 7 (E = 0) X = X on Stack always	FFFE,F BRK bit = 0 on stack if $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$	FFFE,F BRK bit = 0 on stack if $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$
8 Interrupt or Break Bank Address	PBR not pushed (E = 1) RTI PBR not pulled (E = 1) PBR pushed (E = 0) RTI PBR pulled (E = 0)	Not available	Not available
9 Memory Lock ($\overline{\text{ML}}$)	$\overline{\text{ML}}$ = 0 during Read, Modify and Write cycles	$\overline{\text{ML}}$ = 0 during Modify and Write	Not available
10. Indexed Across Page Boundary (d),y, a,x; a,y	Extra read of invalid address (Note 1)	Extra read of last instruction fetch	Extra read of invalid address
11 RDY Pulled During Write Cycle	Ignored (E = 1) for W65C802 only. Processor stops (E = 0)	Processor stops	Ignored
12 WAI and STP Instructions	Available	Available	Not available
13 Unused OP Codes	One reserved OP Code specified as WDM will be used in future systems. The W65C816 performs a no-operation	No operation	Unknown and some "hang up" processor
14 Bank Address Handling	PBR - 00 after reset or interrupts	Not available	Not available
15 R/$\overline{\text{W}}$ During Read-Modify-Write Instructions	E = 1, R/$\overline{\text{W}}$ = 0 during Modify and Write cycles E - 0, R/$\overline{\text{W}}$ - 0 only during Write cycle	R/$\overline{\text{W}}$ – 0 only during Write cycle	R/$\overline{\text{W}}$ = 0 during Modify and Write cycles

16 Pin 7	W65C802 = SYNC
	W65C816 = VPA
		SYNC	SYNC

17 COP Instruction Signatures 00-7F user defined Signatures 80-FF reserved	Available	Not available	Not available

Note 1 See Caveat section for additional information

W65C802 and W65C816 Microprocessor Addressing Modes

The W65C816 is capable of directly addressing 16 MBytes of memory. This address space has special significance within certain addressing modes, as follows:

Reset and Interrupt Vectors

The Reset and Interrupt vectors use the majority of the fixed addresses between 00FFE0 and 00FFFF

Stack

The Stack may use memory from 000000 to 00FFFF. The effective address of Stack and Stack Relative addressing modes will always be within this range.

Direct

The Direct addressing modes are usually used to store memory registers and pointers The effective address generated by Direct, Direct,X and Direct,Y addressing modes is always in Bank 0 (000000-00FFFF)

Program Address Space

The Program Bank register is not affected by the Relative, Relative Long, Absolute, Absolute Indirect, and Absolute Indexed Indirect addressing modes or by incrementing the Program Counter from FFFF. The only instructions that affect the Program Bank register are. RTI, RTL, JML, JSL, and JMP Absolute Long. Program code may exceed 64K bytes although code segments may not span bank boundaries.

Data Address Space

The data address space is contiguous throughout the 16 MByte address space. Words, arrays, records, or any data structures may span 64 KByte bank boundaries with no compromise in code efficiency The following addressing modes generate 24-bit effective addresses

- Direct Indexed Indirect (d,x)
- Direct Indirect Indexed (d),y
- Direct Indirect (d)
- Direct Indirect Long [d]
- Direct Indirect Long Indexed [d],y
- Absolute a
- Absolute a,x
- Absolute a,y
- Absolute Long al
- Absolute Long Indexed al,x
- Stack Relative Indirect Indexed (d,s),y

The following addressing mode descriptions provide additional detail as to how effective addresses are calculated.

Twenty-four addressing modes are available for use with the W65C802 and W65C816 microprocessors. The "long" addressing modes may be used with the W65C802; however, the high byte of the address is not available to the hardware. Detailed descriptions of the 24 addressing modes are as follows

1. Immediate Addressing—#

The operand is the second byte (second and third bytes when in the 16-bit mode) of the instruction

2. Absolute—a

With Absolute addressing the second and third bytes of the instruction form the low-order 16 bits of the effective address. The Data Bank Register contains the high-order 8 bits of the operand address

Instruction:	opcode	addrl	addrh
Operand Address:	DBR	addrh	addrl

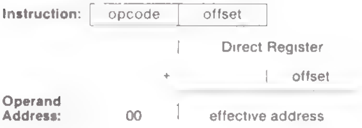
3. Absolute Long—al

The second, third, and fourth byte of the instruction form the 24-bit effective address

Instruction:	opcode	addrl	addrh	baddr
Operand Address:	baddr	addrh	addrl	

4. Direct—d

The second byte of the instruction is added to the Direct Register (D) to form the effective address An additional cycle is required when the Direct Register is not page aligned (DL not equal 0) The Bank register is always 0



5. Accumulator—A

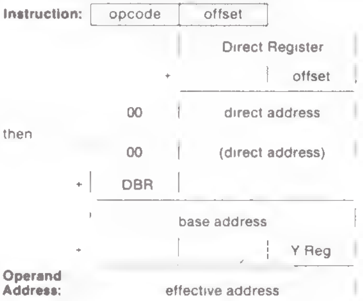
This form of addressing always uses a single byte instruction. The operand is the Accumulator

6. Implied—i

Implied addressing uses a single byte instruction The operand is implicitly defined by the instruction.

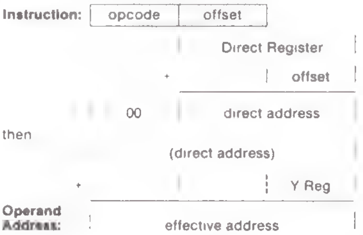
7. Direct Indirect Indexed—(d),y

This address mode is often referred to as Indirect,Y. The second byte of the instruction is added to the Direct Register (D). The 16-bit contents of this memory location is then combined with the Data Bank register to form a 24-bit base address. The Y Index Register is added to the base address to form the effective address



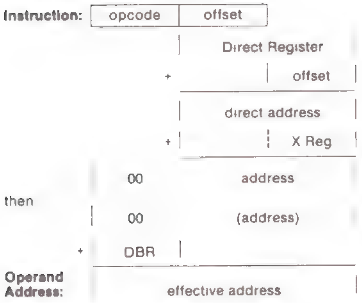
8. Direct Indirect Long Indexed—[d],y

With this addressing mode, the 24-bit base address is pointed to by the sum of the second byte of the instruction and the Direct Register The effective address is this 24-bit base address plus the Y Index Register



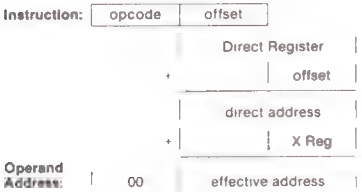
9. Direct Indexed Indirect—(d,x)

This address mode is often referred to as Indirect,X The second byte of the instruction is added to the sum of the Direct Register and the X Index Register. The result points to the low-order 16 bits of the effective address. The Data Bank Register contains the high-order 8 bits of the effective address



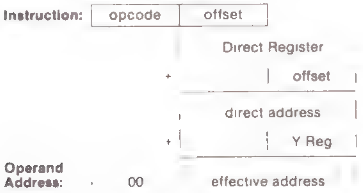
10. Direct Indexed With X—d,x

The second byte of the instruction is added to the sum of the Direct Register and the X Index Register to form the 16-bit effective address. The operand is always in Bank 0



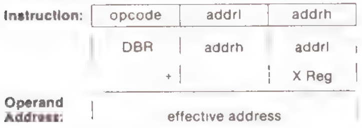
11. Direct Indexed With Y—d,y

The second byte of the instruction is added to the sum of the Direct Register and the Y Index Register to form the 16-bit effective address. The operand is always in Bank 0



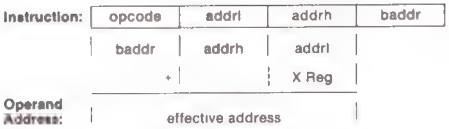
12. Absolute Indexed With X—a,x

The second and third bytes of the instruction are added to the X Index Register to form the low-order 16 bits of the effective address The Data Bank Register contains the high-order 8 bits of the effective address



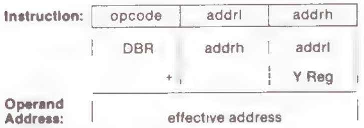
13. Absolute Long Indexed With X—al,x

The second, third and fourth bytes of the instruction form a 24-bit base address. The effective address is the sum of this 24-bit address and the X Index Register



14. Absolute Indexed With Y—a,y

The second and third bytes of the instruction are added to the Y Index Register to form the low-order 16 bits of the effective address. The Data Bank Register contains the high-order 8 bits of the effective address



15. Program Counter Relative—r

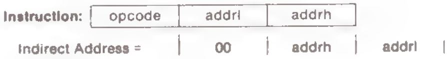
This address mode, referred to as Relative Addressing, is used only with the Branch instructions. If the condition being tested is met, the second byte of the instruction is added to the Program Counter, which has been updated to point to the opcode of the next instruction. The offset is a signed 8-bit quantity in the range from -128 to 127. The Program Bank Register is not affected.

16. Program Counter Relative Long—rl

This address mode, referred to as Relative Long Addressing, is used only with the Unconditional Branch Long instruction (BRL) and the Push Effective Relative instruction (PER). The second and third bytes of the instruction are added to the Program Counter, which has been updated to point to the opcode of the next instruction. With the branch instruction, the Program Counter is loaded with the result. With the Push Effective Relative instruction, the result is stored on the stack. The offset is a signed 16-bit quantity in the range from -32768 to 32767. The Program Bank Register is not affected.

17. Absolute Indirect—(a)

The second and third bytes of the instruction form an address to a pointer in Bank 0. The Program Counter is loaded with the first and second bytes at this pointer. With the Jump Long (JML) instruction, the Program Bank Register is loaded with the third byte of the pointer.



New PC = (indirect address)

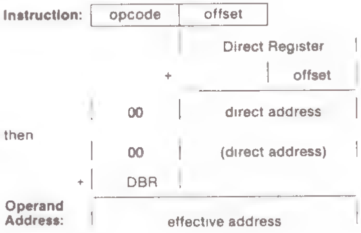
with JML

New PC = (indirect address)

New PBR = (indirect address +2)

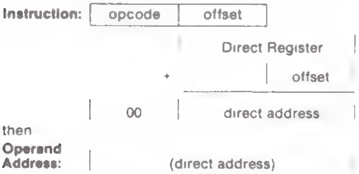
18. Direct Indirect—(d)

The second byte of the instruction is added to the Direct Register to form a pointer to the low-order 16 bits of the effective address. The Data Bank Register contains the high-order 8 bits of the effective address



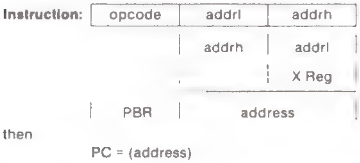
19. Direct Indirect Long—[d]

The second byte of the instruction is added to the Direct Register to form a pointer to the 24-bit effective address.



20. Absolute Indexed Indirect—(a,x)

The second and third bytes of the instruction are added to the X Index Register to form a 16-bit pointer in Bank 0. The contents of this pointer are loaded in the Program Counter. The Program Bank Register is not changed

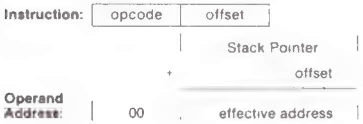


21. Stack—s

Stack addressing refers to all instructions that push or pull data from the stack, such as Push, Pull Jump to Subroutine, Return from Subroutine, Interrupts, and Return from Interrupt. The bank address is always 0. Interrupt Vectors are always fetched from Bank 0.

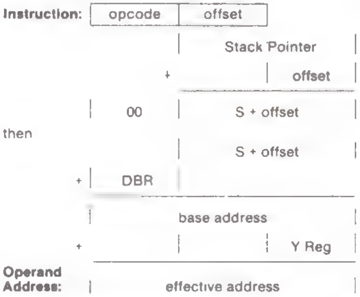
22. Stack Relative—d,s

The low-order 16 bits of the effective address is formed from the sum of the second byte of the instruction and the Stack Pointer. The high-order 8 bits of the effective address is always zero The relative offset is an unsigned 8-bit quantity in the range of 0 to 255.



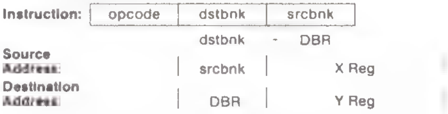
23. Stack Relative Indirect Indexed—(d,s),y

The second byte of the instruction is added to the Stack Pointer to form a pointer to the low-order 16-bit base address in Bank 0. The Data Bank Register contains the high-order 8 bits of the base address. The effective address is the sum of the 24-bit base address and the Y Index Register



24. Block Source Bank, Destination Bank—xyc

This addressing mode is used by the Block Move instructions. The second byte of the instruction contains the high-order 8 bits of the destination address. The Y index Register contains the low-order 16 bits of the destination address. The third byte of the instruction contains the high-order 8 bits of the source address. The X Index Register contains the low-order 16 bits of the source address. The C Accumulator contains one less than the number of bytes to move. The second byte of the block move instructions is also loaded into the Data Bank Register.



Increment (MVN) or decrement (MVP) X and Y.
Decrement C (if greater than zero), then PC+3 → PC.

Table 2. W65C802 and W65C816 Instruction Set—Alphabetical Sequence

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift One Bit Left Memory or Accumulator
BCC	Branch on Carry Clear (P_C = 0)
BCS	Branch on Carry Set (P_C - 1)
BEQ	Branch if Equal (P_Z 1)
BIT	Bit Test
BMI	Branch if Result Minus (P_N 1)
BNE	Branch if Not Equal (P_Z 0)
BPL	Branch if Result Plus (P_N = 0)
BRA	Branch Always
BRK	Force Break
BRL	Branch Always Long
BVC	Branch on Overflow Clear (P_V = 0)
BVS	Branch on Overflow Set (P_V 1)
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
COP	Coprocessor
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory or Accumulator by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive OR" Memory with Accumulator
INC	Increment Memory or Accumulator by One
INX	Increment Index X by One
INY	Increment Index Y by One
JML	Jump Long
JMP	Jump to New Location
JSL	Jump Subroutine Long
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift One Bit Right (Memory or Accumulator)
MVN	Block Move Negative
MVP	Block Move Positive
NOP	No Operation
ORA	"OR" Memory with Accumulator
PEA	Push Effective Absolute Address on Stack (or Push Immediate Data on Stack)
PEI	Push Effective Indirect Address on Stack (or Push Direct Data on Stack)
PER	Push Effective Program Counter Relative Address on Stack
PHA	Push Accumulator on Stack
PHB	Push Data Bank Register on Stack
PHD	Push Direct Register on Stack
PHK	Push Program Bank Register on Stack
PHP	Push Processor Status on Stack
PHX	Push Index X on Stack
PHY	Push Index Y on Stack
PLA	Pull Accumulator from Stack
PLB	Pull Data Bank Register from Stack
PLD	Pull Direct Register from Stack
PLP	Pull Processor Status from Stack
PLX	Pull Index X from Stack
PLY	Pull Index Y form Stack
REP	Reset Status Bits
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTL	Return from Subroutine Long
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
SEP	Set Proce or Status Bite
STA	Store Accumulator in Memory
STP	Stop the Clock
STX	Store Index X in Memory
STY	Store Index Y in Memory
STZ	Store Zero in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TCD	Transfer C Accumulator to Direct Register
TCS	Transfer C Accumulator to Stack Pointer Register
TDC	Transfer Direct Register to C Accumulator
TRB	Test and Reset Bit
TSB	Test and Set Bit
TSC	Transfer Stack Pointer Register to C Accumulator
TSX	Transfer Stack Pointer Register to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer Register
TXY	Transfer Index X to Index Y
TYA	Transfer Index Y to Accumulator
TYX	Transfer Index Y to Index X
WAI	Wait for Interrupt
WDM	Reserved for Future Use
XBA	Exchange B and A Accumulator
XCE	Exchange Carry and Emulation Bits

For alternate mnemonics, see Table 7.

Table 3. Vector Locations

E = 1		E 0	
OOFFFE F —$\overline{\text{IRQ}}$/BRK	Hardware/Software	OOFFEE,F —$\overline{\text{IRQ}}$	Hardware
OOFFFC,D—$\overline{\text{RESET}}$	Hardware	OOFFEC D —(Reserved)	
OOFFFA,B —$\overline{\text{NMI}}$	Hardware	OOFFEA,B —$\overline{\text{NMI}}$	Hardware
OOFFF8,9 —$\overline{\text{ABORT}}$	Hardware	OOFFE8,9 —$\overline{\text{ABORT}}$	Hardware
OOFFF6,7 —(Reserved)		OOFFE6,7 —BRK	Software
OOFFF4,5 —COP	Software	OOFFE4,5 —COP	Software

The VP output is low during the two cycles used for vector location access
When an interrupt is executed, D = 0 and I = 1 in Status Register P

Table 4. Opcode Matrix

MSD	LSD 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	MSD
0	BRK s 2 8	ORA (d,x) 2 6	COP s 2 * 8	ORA d,s 2 * 4	TSB d 2 ● 5	ORA d 2 3	ASL d 2 5	ORA [d] 2 * 6	PHP s 1 3	ORA # 2 2	ASL A 1 2	PHD s 1 * 4	TSB a 3 ● 6	ORA a 3 4	ASL a 3 6	ORA al 4 * 5	0
1	BPL r 2 2	ORA (d),y 2 5	ORA (d) 2 ● 5	ORA (d,s),y 2 * 7	TRB d 2 ● 5	ORA d,x 2 4	ASL d,x 2 6	ORA [d],y 2 * 6	CLC i 1 2	ORA a,y 3 4	INC A 1 ● 2	TCS i 1 * 2	TRB a 3 ● 6	ORA a,x 3 4	ASL a,x 3 7	ORA al,x 4 * 5	1
2	JSR a 3 6	AND (d,x) 2 6	JSL al 4 * 8	AND d,s 2 * 4	BIT d 2 3	AND d 2 3	ROL d 2 5	AND [d] 2 * 6	PLP s 1 4	AND # 2 2	ROL A 1 2	PLD s 1 * 5	BIT a 3 4	AND a 3 4	ROL a 3 6	AND al 4 * 5	2
3	BMI r 2 2	AND (d),y 2 5	AND (d) 2 ● 5	AND (d,s),y 2 * 7	BIT d,x 2 ● 4	AND d,x 2 4	ROL d,x 2 6	AND [d],y 2 * 6	SEC i 1 2	AND a,y 3 4	DEC A 1 ● 2	TSC i 1 * 2	BIT a,x 3 ● 4	AND a,x 3 4	ROL a,x 3 7	AND al,x 4 * 5	3
4	RTI s 1 7	EOR (d,x) 2 6	WDM 2 * 2	EOR d,s 2 * 4	MVP xyc 3 * 7	EOR d 2 3	LSR d 2 5	EOR [d] 2 * 6	PHA s 1 3	EOR # 2 2	LSR A 1 2	PHK s 1 * 3	JMP a 3 3	EOR a 3 4	LSR a 3 6	EOR al 4 * 5	4
5	BVC r 2 2	EOR (d),y 2 5	EOR (d) 2 ● 5	EOR (d,s),y 2 * 7	MVN xyc 3 * 7	EOR d,x 2 4	LSR d,x 2 6	EOR [d],y 2 * 6	CLI i 1 2	EOR a,y 3 4	PHY s 1 ● 3	TCD i 1 * 2	JMP al 4 * 4	EOR a,x 3 4	LSR a,x 3 7	EOR al,x 4 * 5	5
6	RTS s 1 6	ADC (d,x) 2 6	PER s 3 * 6	ADC d,s 2 * 4	STZ d 2 ● 3	ADC d 2 3	ROR d 2 5	ADC [d] 2 * 6	PLA s 1 4	ADC # 2 2	ROR A 1 2	RTL s 1 * 6	JMP (a) 3 5	ADC a 3 4	ROR a 3 6	ADC al 4 * 5	6
7	BVS r 2 2	ADC (d) y 2 5	ADC (d) 2 ● 5	ADC (d,s),y 2 * 7	STZ d,x 2 ● 4	ADC d,x 2 4	ROR d,x 2 6	ADC [d],y 2 * 6	SEI i 1 2	ADC a,y 3 4	PLY s 1 ● 4	TDC i 1 * 2	JMP (a,x) 3 ● 6	ADC a,x 3 4	ROR a,x 3 7	ADC al,x 4 * 5	7
8	BRA r 2 ● 2	STA (d,x) 2 6	BRL rl 3 * 3	STA d,s 2 * 4	STY d 2 3	STA d 2 3	STX d 2 3	STA [d] 2 * 6	DEY i 1 2	BIT # 2 ● 2	TXA i 1 2	PHB s 1 * 3	STY a 3 4	STA a 3 4	STX a 3 4	STA al 4 * 5	8
9	BCC r 2 2	STA (d),y 2 6	STA (d) 2 ● 5	STA (d,s),y 2 * 7	STY d x 2 4	STA d,x 2 4	STX d y 2 4	STA [d],y 2 * 6	TYA i 1 2	STA a,y 3 5	TXS i 1 2	TXY i 1 * 2	STZ a 3 ● 4	STA a,x 3 5	STZ a,x 3 ● 5	STA al,x 4 * 5	9
A	LDY # 2 2	LDA (d,x) 2 6	LDX # 2 2	LDA d,s 2 * 4	LDY d 2 3	LDA d 2 3	LDX d 2 3	LDA [d] 2 * 6	TAY i 1 2	LDA # 2 2	TAX i 1 2	PLB s 1 * 4	LDY a 3 4	LDA a 3 4	LDX a 3 4	LDA al 4 * 5	A
B	BCS r 2 2	LDA (d),y 2 5	LDA (d) 2 ● 5	LDA (d,s),y 2 * 7	LDY d,x 2 4	LDA d,x 2 4	LDX d,y 2 4	LDA [d],y 2 * 6	CLV i 1 2	LDA a,y 3 4	TSX i 1 2	TYX i 1 * 2	LDY a,x 3 4	LDA a x 3 4	LDX a,y 3 4	LDA al,x 4 * 5	B
C	CPY # 2 2	CMP (d,x) 2 6	REP # 2 * 3	CMP d,s 2 * 4	CPY d 2 3	CMP d 2 3	DEC d 2 5	CMP [d] 2 * 6	INY i 1 2	CMP # 2 2	DEX i 1 2	WAI i 1 ● 3	CPY a 3 4	CMP a 3 4	DEC a 3 6	CMP al 4 * 5	C
D	BNE r 2 2	CMP (d),y 2 5	CMP (d) 2 ● 5	CMP (d,s),y 2 * 7	PEI s 2 * 6	CMP d,x 2 4	DEC d,x 2 6	CMP [d],y 2 * 6	CLD i 1 2	CMP a,y 3 4	PHX s 1 ● 3	STP i 1 ● 3	JML (a) 3 * 6	CMP a,x 3 4	DEC a,x 3 7	CMP al,x 4 * 5	D
E	CPX # 2 2	SBC (d,x) 2 6	SEP # 2 * 3	SBC d,s 2 * 4	CPX d 2 3	SBC d 2 3	INC d 2 5	SBC [d] 2 * 6	INX i 1 2	SBC # 2 2	NOP i 1 2	XBA i 1 * 3	CPX a 3 4	SBC a 3 4	INC a 3 6	SBC al 4 * 5	E
F	BEQ r 2 2	SBC (d),y 2 5	SBC (d) 2 ● 5	SBC (d,s) y 2 * 7	PEA s 3 * 5	SBC d,x 2 4	INC d,x 2 6	SBC [d],y 2 * 6	SED i 1 2	SBC a,y 3 4	PLX s 1 ● 4	XCE i 1 * 2	JSR (a,x) 3 * 6	SBC a,x 3 4	INC a,x 3 7	SBC al,x 4 * 5	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

symbol	addressing mode	symbol	addressing mode
#	immediate	[d]	direct indirect long
A	accumulator	[d],y	direct indirect long indexed
r	program counter relative	a	absolute
rl	program counter relative long	a,x	absolute indexed (with x)
i	implied	a,y	absolute indexed (with y)
s	stack	al	absolute long
d	direct	al,x	absolute long indexed
d,x	direct indexed (with x)	d,s	stack relative
d,y	direct indexed (with y)	(d,s),y	stack relative indirect indexed
(d)	direct indirect	(a)	absolute indirect
(d,x)	direct indexed indirect	(a,x)	absolute indexed indirect
(d),y	direct indirect indexed	xyc	block move

Op Code Matrix Legend

INSTRUCTION MNEMONIC	* = New W65C816/802 Opcodes ● - New W65C02 Opcodes Blank = NMOS 6502 Opcodes	ADDRESSING MODE
BASE NO BYTES		BASE NO. CYCLES

Table 5. Operation, Operation Codes, and Status Register

MNE-MONIC		#	a	al	d	A	i	(d),y	[d],y	(d,x)	d,x	d,y	a,x	al,x	a,y	r	rl	(a)	(d)	[d]	(a,x)	s	d,s	(d,s),y	xyc	PROCESSOR STATUS CODE									MNE-MONIC
																										7	6	5	4	3	2	1	0		
																										N	V	M	X	D	I	Z	C		E = 0
	OPERATION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	N	V	1	B	D	I	Z	C		E = 1
ADC	A + M + C → A	69	6D	6F	65			71	77	61	75		7D	7F	79				72	67			63	73		N	V					Z	C		ADC
AND	A∧M → A	29	2D	2F	25			31	37	21	35		3D	3F	39				32	27			23	33		N						Z			AND
ASL	C ← [15/7 0] ← 0		0E		06	0A					16		1E													N						Z	C		ASL
BCC	BRANCH IF C = 0															90																			BCC
BCS	BRANCH IF C = 1															B0																			BCS
BEQ	BRANCH IF Z = 1															F0																			BEQ
BIT	A∧M (NOTE 1)	89	2C		24						34		3C													M_7	M_6					Z			BIT
BMI	BRANCH IF N = 1															30																			BMI
BNE	BRANCH IF Z = 0															D0																			BNE
BPL	BRANCH IF N = 0															10																			BPL
BRA	BRANCH ALWAYS															80																		●	BRA
BRK	BREAK (NOTE 2)																					00							●	0	1				BRK
BRL	BRANCH LONG ALWAYS																82																	★	BRL
BVC	BRANCH IF V = 0															50																			BVC
BVS	BRANCH IF V = 1															70																			BVS
CLC	0 → C						18																										0		CLC
CLD	0 → D						D8																							0					CLD
CLI	0 → I						58																								0				CLI
CLV	0 → V						B8																				0								CLV
CMP	A - M	C9	CD	CF	C5			D1	D7	C1	D5		DD	DF	D9				D2	C7			C3	D3		N						Z	C		CMP
COP	CO-PROCESSOR																					02								0	1			★	COP
CPX	X - M	E0	EC		E4																					N						Z	C		CPX
CPY	Y - M	C0	CC		C4																					N						Z	C		CPY
DEC	DECREMENT		CE		C6	3A					D6		DE													N						Z			DEC
DEX	X - 1 → X						CA																			N						Z			DEX
DEY	Y - 1 → Y						88																			N						Z			DEY
EOR	A⊻M → A	49	4D	4F	45			51	57	41	55		5D	5F	59				52	47			43	53		N						Z			EOR
INC	INCREMENTS		EE		E6	1A					F6		FE													N						Z			INC
INX	X + 1 → X						E8																			N						Z			INX
INY	Y + 1 → Y						C8																			N						Z			INY
JML	JUMP LONG TO NEW LOC																	DC																★	JML
JMP	JUMP TO NEW LOC		4C	5C														6C			7C														JMP
JSL	JUMP LONG TO SUB			22																															JSL
JSR	JUMP TO SUB		20																		FC														JSR
LDA	M → A	A9	AD	AF	A5			B1	B7	A1	B5		BD	BF	B9				B2	A7			A3	B3		N						Z			LDA
LDX	M → X	A2	AE		A6							B6			BE											N						Z			LDX
LDY	M → Y	A0	AC		A4						B4		BC													N						Z			LDY
LSR	0 → [15/7 0] → C		4E		46	4A					56		5E													0						Z	C		LSR
MVN	M → M NEGATIVE																								54									★	MVN
MVP	M → M POSITIVE																								44									★	MVP
NOP	NO OPERATION						EA																												NOP
ORA	A∨M → A	09	0D	0F	05			11	17	01	15		1D	1F	19				12	07			03	13		N						Z			ORA
PEA	Mpc + 1, Mpc + 2 → Ms - 1, Ms S - 2 → S																					F4												★	PEA
PEI	M(d), M(d + 1) → Ms - 1, Ms S - 2 → S																					D4												★	PEI
PER	Mpc + rl, Mpc + rl + 1 → Ms - 1, Ms S - 2 → S																					62												★	PER
PHA	A → Ms, S - 1 → S																					48													PHA
PHB	DBR → Ms, S - 1 → S																					8B													PHB
PHD	D → Ms, Ms - 1, S - 2 → S																					0B												★	PHD
PHK	PBR → Ms, S - 1 → S																					4B												★	PHK
PHP	P → Ms, S - 1 → S																					08													PHP
PHX	X → Ms, S - 1 → S																					DA												●	PHX
PHY	Y → Ms, S - 1 → S																					5A												●	PHY
PLA	S + 1 → S, Ms → A																					68				N						Z			PLA
PLB	S + 1 → S, Ms → DBR																					AB				N						Z		★	PLB
PLD	S + 2 → S, Ms - 1, Ms → D																					2B				N						Z		★	PLD
PLP	S + 1 → S, Ms → P																					28				N	V	M	X	D	I	Z	C		PLP
PLX	S + 1 → S, Ms → X																					FA				N						Z			PLX
PLY	S + 1 → S, Ms → Y																					7A				N						Z			PLY
REP	M∧P → P	C2																								N	V	M	X	D	I	Z	C	★	REP
ROL	[15/7 0] ← C		2E		26	2A					36		3E													N						Z	C		ROL
ROR	C → [15/7 0]		6E		66	6A					76		7E													N						Z	C		ROR
RTI	RTRN FROM INT																					40				N	V	M	X	D	I	Z	C		RTI
RTL	RTRN FROM SUB LONG																					[illegible]												★	RTL
RTS	RTRN SUBROUTINE																					60													RTS
SBC	A - M - C̄ → A	E9	ED	EF	E5			F1	F7	E1	F5		FD	FF	F9				F2	E7			E3	F3		N	V					Z	C		SBC
SEC	1 → C						38																										1		SEC
SED	1 → D						F8																							1					SED
SEI	1 → I						78																								1				SEI
SEP	M∨P → P	E2																								N	V	M	X	D	I	Z	C	★	SEP
STA	A → M		8D	8F	85			91	97	81	95		9D	9F	99				92	87			83	93											STA
STP	STOP (1 → φ2)						DB																											●	STP
STX	X → M		8E		86							96																							STX
STY	Y → M		8C		84						94																								STY
STZ	00 → M		9C		64						74		9E																					●	STZ
TAX	A → X						AA																			N						Z			TAX
TAY	A → Y						A8																			N						Z			TAY
TCD	C → D						5B																			N						Z		★	TCD
TCS	C → S						1B																											★	TCS
TDC	D → C						7B																			N						Z		★	TDC
TRB			1C		14																											Z		●	TRB
TSB	A∨M → M		0C		04																											Z		●	TSB
TSC	S → C						3B																			N						Z		★	TSC
TSX	S → X						BA																			N						Z			TSX
TXA	X → A						8A																			N						Z			TXA
TXS	X → S						9A																												TXS
TXY	X → Y						9B																			N						Z		★	TXY
TYA	Y → A						98																			N						Z			TYA
TYX	Y → X						BB																			N						Z		★	TYX
WAI	0 → RDY						CB																											●	WAI
WDM	NO OPERATION (RESERVED)						42																											★	WDM
XBA	B ↔ A						EB																			N						Z		★	XBA
XCE	C ↔ E						FB																										E	★	XCE

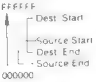
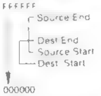
Notes
1 Bit immediate N and V flags not affected. When M = 0, M_{15} → N and M_{14} → V
2 Break Bit (B) in Status register indicates hardware or software break
3 ★ New W65C816/802 Instructions
● New W65C02 Instructions
Blank NMOS 6502

+ Add
- Subtract
∧ AND
V OR
⊻ Exclusive OR

Table 6. Detailed Instruction Operation

	ADDRESS MODE		CYCLE	$\overline{VP}$,	$\overline{ML}$,	VDA,	VPA		ADDRESS BUS	DATA BUS	R/$\overline{W}$
1	Immediate #		1	1	1	1	1		PBR PC	Op Code	1
	(LDY CPY CPX,LDX ORA		2	1	1	0	1		PBR PC+1	IDL	1
	AND EOR ADC BIT LDA	(1)(8)	2a	1	1	0	1		PBR PC+2	IDH	1
	CMP SBC REP SEP)										
	(14 Op Codes)										
	(2 and 3 bytes)										
	(2 and 3 cycles)										
2a	Absolute a		1	1	1	1	1		PBR PC	Op Code	1
	(BIT,STY,STZ LDY		2	1	1	0	1		PBR PC+1	AAL	1
	CPY CPX STX LDX		3	1	1	0	1		PBR PC+2	AAH	1
	ORA AND EOR ADC		4	1	1	1	0		DBR AA	Data Low	1/0
	STA LDA CMP SBC)	(1)	4a	1	1	1	0		DBR AA+1	Data High	1/0
	(18 Op Codes)										
	(3 bytes)										
	(4 and 5 cycles)										
2b	Absolute (R M W) a		1		1	1	1		PBR PC	Op Code	1
			2	1	1	0	1		PBR PC+1	AAL	1
	(ASL ROL LSR ROR		3	1	1	0	1		PBR PC+2	AAH	1
	DEC INC,TSB,TRB)		4	1	0	1	0		DBR AA	Data Low	1
	(6 Op Codes)	(1)	4a	1	0	1	0		DBR AA+1	Data High	1
	(3 bytes)	(3)	5	1	0	0	0		DBR AA+1	IO	1
	(6 and 8 cycles)	(1)	6a	1	0	1	0		DBR AA+1	Data High	0
			6	1	0	1	0		DBR AA	Data Low	0
2c	Absolute (JUMP) a		1	1	1	1	1		PBR PC	Op Code	1
	(JMP),(4C)		2	1	1	0	1		PBR PC+1	NEW PCL	1
	(1 Op Code)		3	1	1	0	1		PBR PC+2	NEW PCH	1
	(3 bytes)		1	1	1	1	1		PBR NEW PC	Op Code	1
	(3 cycles)										
2d	Absolute (Jump to		1	1	1	1	1		PBR PC	Op Code	1
	subroutine) a		2	1	1	0	1		PBR PC+1	NEW PCL	1
	(JSR)		3	1	1	0	1		PBR PC+2	NEW PCH	1
	(1 Op Code)		4	1	1	0	0		PBR PC+2	IO	1
	(3 bytes)		5	1	1	1	0		0,S	PCH	0
	(6 cycles)		6	1	1	1	0		0,S-1	PCL	0
	(different order from N6502)		1	1	1	1	1		PBR NEW PC	Next Op Code	1
★3a	Absolute Long al		1	1	1	1	1		PBR PC	Op Code	1
	(ORA AND EOR ADC		2	1	1	0	1		PBR PC+1	AAL	1
	STA LDA CMP SBC		3	1	1	0	1		PBR PC+2	AAH	1
	(8 Op Codes)		4	1	1	0	1		PBR PC+3	AAB	1
	(4 bytes)		5	1	1	1	0		AAB AA	Data Low	1/0
	(5 and 6 cycles)	(1)	5a	1	1	1	0		AAB AA+1	Data High	1/0
★3b	Absolute Long (JUMP) al		1	1	1	1	1		PBR PC	Op Code	1
	(JMP)		2	1	1	0	1		PBR PC+1	NEW PCL	1
	(1 Op Code)		3	1	1	0	1		PBR PC+2	NEW PCH	1
	(4 bytes)		4	1	1	0	1		PBR PC+3	NEW BR	1
	(4 cycles)		1	1	1	1	1		NEW PBR PC	Op Code	
											1
★3c	Absolute Long (Jump to			1	1	1	1		PBR PC	Op Code	1
	Subroutine Long) al		2	1	1	0	1		PBR PC+1	NEW PCL	
	(JSL)		3	1	1	0	1		PBR PC+2	NEW PCH	1
	(1 Op Code)		4	1	1	1	0		0,S	PBR	0
	(4 bytes)		5	1	1	0	0		0,S	IO	1
	(7 cycles)		6	1	1	0	1		PBR PC+3	NEW PBR	1
			7	1	1	1	0		0,S-1	PCH	0
			8	1	1	1	0		0,S-2	PCL	0
			1	1	1	1	1		NEW PBR PC	Next Op Code	1
4a	Direct d		1	1	1	1	1		PBR PC	Op Code	1
	(BIT STZ STY LDY		2	1	1	0	1		PBR PC+1	DO	1
	CPY CPX STX LDX	(2)	2a	1	1	0	0		PBR PC+1	IO	1
	ORA AND EOR ADC		3	1	1	1	0		0,D+DO	Data Low	1/0
	STA LDA CMP SBC)	(1)	3a	1	1	1	0		0,D+DO+1	Data High	1/0
	(18 Op Codes)										
	(2 bytes)										
	(3 4 and 5 cycles)										
4b	Direct (R M W) d		1	1	1	1	1		PBR PC	Op Code	1
	(ASL ROL,LSR ROR		2		1	0	1		PBR PC+1	DO	1
	DEC INC TSB TRB)	(2)	2a	1	1	0	0		PBR PC+1	IO	1
	(6 Op Codes)		3	1	0	1	0		0,D+DO	Data Low	1
	(2 bytes)	(1)	3a	1	0	1	0		0,D+DO+1	Data High	1
	(5 6 7 and 8 cycles)	(3)	4	1	0	0	0		0,D+DO+1	IO	1
		(1)	5a	1	0	1	0		0,D+DO+1	Data High	0
			5	1	0	1	0		0,D+DO	Data Low	0
5	Accumulator A		1			1	1		PBR PC	Op Code	1
	(ASL INC ROL DEC LSR ROR		2	1	1	0	0		PBR PC+1	IO	1
	(6 Op Codes)										
	(1 byte)										
	(2 cycles)										
6a	Implied i		1	1	1		1		PBR PC	Op Code	1
	(DEY INY INX DEX NOP		2	1	1	0	0		PBR PC+1	IO	1
	XCE TYA TAY TXA TXS										
	TAX TSX TCS TSC TCD										
	TDC TXY TYX CLC SEC										
	CLI SEI CLV CLD SED)										
	(25 Op Codes)										
	(1 byte)										
	(2 cycles)										
★6b	Implied i		1	1		1	1		PBR PC	Op Code	1
	XBA		2	1	1	0	0		PBR PC+1	IO	1
	(1 Op Code)		3	1	1	0	0		PBR PC+1	IO	1
	(1 byte)										
	(3 cycles)										
								RDY			
● 6c	Wait For Interrupt										
	(WAI)		1		1		1	1	PBR PC	Op Code	
	(1 Op Code)	(9)	2			0	0	1	PBR PC+1	IO	1
	(1 byte)		3			0	0	0	PBR PC+1	IO	1
	(3 cycles)	IRQ NMI	1	1	1		1	1	PBR PC+1	IRQ,BRK	1
● 6d	Stop The Clock										
	(STP)		1	1	1	1		1	PBR PC	Op Code	1
	(1 Op Code)		2	1		0	0	1	PBR PC+1	IO	1
	(1 byte)	RES 1	3		1	0	0	1	PBR PC+1	IO	1
	(3 cycles)	$\overline{\text{RES}}$ 0	1c	1	1	0	0	1	PBR PC+1	RES (BRK)	1
		$\overline{\text{RES}}$ 0	1b	1		0	0	1	PBR PC+1	RES,(BRK)	
		RES 1	1a	1	1	0	0	1	PBR PC+1	RES,(BRK)	
	See 21a Stack (Hardware interrupt)		1	1	1	1	1	1	PBR PC+1	BEGIN	1
7	Direct Indirect Indexed (d),y		1	1	1	1	1		PBR PC	Op Code	1
	(ORA,AND EOR ADC		2	1	1	0	1		PBR PC+1	DO	1
	STA,LDA CMP SBC)	(2)	2a	1	1	0	0		PBR PC+1	IO	1
	(8 Op Codes)		3	1	1	1	0		0 D+DO	AAL	1
	(2 bytes)		4	1	1	1	0		0 D+DO+1	AAH	1
	(5 6,7 and 8 cycles)	(4)	4a	1	1	0	0		DBR AAH AAL+ YL	IO	1
			5	1	1	1	0		DBR,AA+Y	Data Low	1/0
		(1)	5a	1	1	1	0		DBR,AA+Y+1	Data High	1/0
8	Direct Indirect		1	1	1	1	1		PBR PC	Op Code	1
	Indexed Long [d],y		2	1	1	0	1		PBR,PC+1	DO	1
	(ORA AND EOR ADC	(2)	2a	1	1	0	0		PBR,PC+1	IO	1
	STA LDA,CMP SBC)		3	1	1	1	0		0,D+DO	AAL	1
	(8 Op Codes)		4	1	1	1	0		0 D+DO+1	AAH	1
	(2 bytes)		5	1	1	1	0		0 D+DO+2	AAB	1
	(6 7 and 8 cycles)		6	1	1	1	0		AAB,AA+Y	Data Low	1/0
		(1)	6a	1	1	1	[illegible]		AAB,AA+Y+1	Data High	1/0
9	Direct Indexed Indirect (d,x)		1	1	1	1	1		PBR PC	Op Code	1
	(ORA AND EOR ADC		2	1	1	0	1		PBR PC+1	DO	1
	STA LDA CMP SBC)	(2)	2a	1	1	0	0		PBR PC+1	IO	1
	(8 Op Codes)		3	1	1	0	0		PBR PC+1	IO	1
	(2 bytes)		4	1	1	1	0		0 D+DO+X	AAL	1
	(6 7 and 8 cycles)		5	1	1	1	0		0 D+DO+X+1	AAH	1
			6	1	1	1	0		DBR AA	Data Low	1/0
		(1)	6a	1	1	1	0		DBR AA+1	Data High	1/0
10a	Direct X d,x		1	1	1	1	1		PBR PC	Op Code	1
	(BIT STZ STY LDY		2	1	1	0	1		PBR,PC+1	DO	1
	ORA AND EOR ADC	(2)	2a	1	1	0	0		PBR,PC+1	IO	1
	STA,LDA,CMP SBC)		3	1	1	0	0		PBR PC+1	IO	1
	(11 Op Codes)		4	1	1	1	0		0 D+DO+X	Data Low	1/0
	(2 bytes)	(1)	4a	1	1	1	0		0 D+DO+X+1	Data High	1/0
	(4 5 and 6 cycles)										
10b	Direct X (R M W) d,x		1	1	1	1	1		PBR PC	Op Code	1
	(ASL,ROL LSR ROR		2	1	1	0	1		PBR PC+1	DO	1
	DEC INC)	(2)	2a	1	1	0	0		PBR PC+1	IO	1
	(6 Op Codes)		3	1	1	0	0		PBR PC+1	IO	1
	(2 bytes)		4	1	0	1	0		0 D+DO+X	Data Low	1
	(6 7 8 and 9 cycles)	(1)	4a	1	0	1	0		0 D+DO+X+1	Data High	1
		(3)	5	1	0	0	0		0 D+DO+X+1	IO	1
		(1)	6a	1	0	1	0		0 D+DO+X+1	Data High	0
			6	1	0	1	0		0 D+DO+X	Data Low	0
11	Direct Y d,y		1	1	1	1	1		PBR PC	Op Code	1
	(STX LDX)		2	1	1	0	1		PBR PC+1	DO	1
	(2 Op Codes)	(2)	2a	1	1	0	0		PBR PC+1	IO	1
	(2 bytes)		3	1	1	0	0		PBR PC+1	IO	1
	(4 5 and 6 cycles)		4	1	1	1	0		0 D+DO+Y	Data Low	1/0
		(1)	4a	1	1	1	0		0 D+DO+Y+1	Data High	1/0
12a	Absolute X a,x		1	1	1	1	1		PBR PC	Op Code	1
	(BIT LDY STZ		2	1	1	0	1		PBR PC+1	AAL	1
	ORA AND EOR ADC		3	1	1	0	1		PBR PC+2	AAH	1
	STA LDA CMP SBC	(4)	3a	1	1	0	0		DBR AAH AAL+XL	IO	1
	(11 Op Codes)		4	1	1	1	0		DBR AA+X	Data Low	1/0
	(3 bytes)	(1)	4a	1	1	1	0		DBR AA+X+1	Data High	1/0
	(4 5 and 6 cycles)										
12b	Absolute X (R M W) a,x		1	1	1	1	1		PBR PC	Op Code	1
	(ASL ROL LSR ROR		2	1	1	0	1		PBR PC+1	AAL	1
	DEC INC)		3	1	1	0	1		PBR PC+2	AAH	1
	(6 Op Codes)		4	1	1	0	0		DBR AAH AAL+XL	IO	1
	(3 bytes)		5	1	0	1	0		DBR AA+X	Data Low	1
	(7 and 9 cycles)	(1)	5a	1	0	1	0		DBR AA+X+1	Data High	1
		(3)	6	1	0	0	0		DBR AA+X+1	IO	1
		(1)	7a	1	0	1	0		DBR AA+X+1	Data High	0
			7	1	0	1	0		DBR AA+X	Data Low	0
★13	Absolute Long X al,x		1	1	1	1	1		PBR PC	Op Code	1
	(ORA AND EOR ADC		2	1	1	0	1		PBR PC+1	AAL	1
	STA LDA CMP SBC		3	1	1	0	1		PBR PC+2	AAH	1
	(8 Op Codes)		4	1	1	0	1		PBR PC+3	AAB	1
	(4 bytes)		5	1	1	1	0		AAB AA+X	Data Low	1/0
	(5 and 6 cycles)	(1)	5a	1	1	1	0		AAB AA+X+1	Data High	1/0
14	Absolute Y a,y		1	1	1	1	1		PBR PC	Op Code	1
	(LDX ORA AND EOR ADC		2	1	1	0	1		PBR PC+1	AAL	1
	STA LDA CMP SBC)		3	1	1	0	1		PBR PC+2	AAH	1
	(9 Op Codes)	(4)	3a	1	1	0	0		DBR AAH AAL+YL	IO	1
	(3 bytes)		4	1	1	1	0		DBR AA+Y	Data Low	1/0
	(4 5 and 6 cycles)	(1)	4a	1	1	1	0		DBR AA+Y+1	Data High	1/0
15	Relative r		1	1	1	1	1		PBR PC	Op Code	1
	(BPL BMI BVC BVS BCC		2	1	1	0	1		PBR PC+1	Offset	1
	BCS BNE BEQ BRA	(5)	2a	1	1	0	0		PBR PC+1	IO	1
	(9 Op Codes)	(6)	2b	1	1	0	0		PBR PC+1	IO	1
	(2 bytes)		1	1	1	1	1		PBR PC+Offset	Op Code	1
	(2 3 and 4 cycles)										
★16	Relative Long rl		1	1	1	1	1		PBR PC	Op Code	1
	(BRL)		2	1	1	0	1		PBR PC+1	Offset Low	1
	(1 Op Code)		3	1	1	0	1		PBR PC+2	Offset High	1
	(3 bytes)		4	1	1	0	0		PBR PC+2	IO	1
	(4 cycles)		1	1	1	1	1		PBR PC+Offset	Op Code	1
17a	Absolute Indirect (a)		1	1	1	1	1		PBR PC	Op Code	1
	(JMP)		2	1	1	0	1		PBR PC+1	AAL	1
	(1 Op Code)		3	1	1	0	1		PBR PC+2	AAH	1
	(3 bytes)		4	1	1	1	0		0 AA	NEW PCL	1
	(5 cycles)		5	1	1	1	0		0 AA+1	NEW PCH	1
				1	1	1	1		PBR NEW PC	Op Code	1
★17b	Absolute Indirect (a)		1	1	1	1	1		PBR PC	Op Code	1
			2	1	1	0	1		PBR PC+1	AAL	1
	(JML)		3	1	1	0	1		PBR PC+2	AAH	1
	(1 Op Code)		4	1	1	1	0		0 AA	NEW PCL	1
	(3 bytes)		5	1	1	1	0		0 AA+1	NEW PCH	1
	(6 cycles)		6	1	1	1	0		0 AA+2	NEW PBR	1
			1	1	1	1	1		NEW PBR PC	Op Code	1
● 18	Direct Indirect (d)		1	1	1	1	1		PBR PC	Op Code	1
	(ORA AND EOR ADC		2	1	1	0	1		PBR PC+1	DO	1
	STA LDA CMP SBC	(2)	2a	1	1	0	0		PBR PC+1	IO	1
	(8 Op Codes)		3	1		1	0		0 D+DO	AAL	1
	(2 bytes)		4	1	1	1	0		0 D+DO+1	AAH	1
	(5 6 and 7 cycles)		5	1		1	0		DBR AA	Data Low	1/0
		(1)	5a	1	1	1	0		DBR AA+1	Data Low	1/0

Table 6. Detailed Instruction Operation (continued)

	ADDRESS MODE		CYCLE	VP	$\overline{ML}$	VDA	VPA	ADDRESS BUS	DATA BUS	R/$\overline{W}$
★19	Direct Indirect Long [d] (ORA AND EOR ADC STA LDA CMP SBC) (8 Op Codes) (2 bytes) (6 7 and 8 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	DO	1
		(2)	2a	1	1	0	0	PBR,PC+1	IO	1
			3	1	1	1	0	0,D+DO	AAL	1
			4	1	1	1	0	0,D+DO+1	AAH	1
			5	1	1	1	0	0,D+DO+2	AAB	1
			6	1	1	1	0	AAB,AA	Data Low	1/0
		(1)	6a	1	1	1	0	AAB,AA+1	Data High	1/0
20a	Absolute Indexed Indirect (a,x) (JMP) (1 Op Code) (3 bytes) (6 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	AAL	1
			3	1	1	0	1	PBR,PC+2	AAH	1
			4	1	1	0	0	PBR,PC+2	IO	1
			5	1	1	0	1	PBR,AA+X	NEW PCL	1
			6	1	1	0	1	PBR,AA+X+1	NEW PCH	1
			1	1	1		1	PBR,NEW PC	Op Code	
★20b	Absolute Indexed Indirect (Jump to Subroutine Indexed Indirect) (a,x) (JSR) (1 Op Code) (3 bytes) (8 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	AAL	1
			3	1	1	1	0	0,S	PCH	0
			4	1	1	1	0	0,S-1	PCL	0
			5	1	1	0	1	PBR,PC+2	AAH	1
			6	1	1	0	0	PBR,PC+2	IO	1
			7	1	1	0	1	PBR,AA+X	NEW PCL	1
			8	1	1	0	1	PBR,AA+X+1	NEW PCH	1
			1	1	1	1	1	PBR,NEW PC	Next Op Code	1
21a	Stack (Hardware Interrupts) s (IRQ NMI ABORT RES) (4 hardware interrupts) (0 bytes) (7 and 8 cycles)		1	1	1	1	1	PBR,PC	IO	1
		(3)	2	1	1	0	0	PBR,PC	IO	1
		(7)	3	1	1	1	0	0,S	PBR	0
			4	1	1	1	0	0,S-1	PCH	0
			5	1	1	1	0	0,S-2	PCL	0
			6	1	1	1	0	0,S-3	P	0
			7	0	1	1	0	0,VA	AAVL	1
			8	0	1	1	0	0,VA+1	AAVH	1
			1	1	1	1	1	0,AAV	Next Op Code	1
21b	Stack (Software Interrupts) s (BRK COP) (2 Op Codes) (2 bytes) (7 and 8 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
		(3)	2	1	1	0	1	PBR,PC+1	Signature	1
		(7)	3	1	1	1	0	0,S	PBR	0
		(10)	4	1	1	1	0	0,S-1	PCH	0
		(10)	5	1	1	1	0	0,S-2	PCL	0
		(10)	6	1	1	1	0	0,S-3 (COP Latches)	P	0
			7	0	1	1	0	0,VA	AAVL	1
			8	0	1	1	0	0,VA+1	AAVH	1
			1	1	1	1	1	0,AAV	Next Op Code	1
21c	Stack (Return from Interrupt) s (RTI) (1 Op Code) (1 byte) (6 and 7 cycles) (different order from N6502)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	0	PBR,PC+1	IO	1
		(3)	3	1	1	0	0	PBR,PC+1	IO	1
			4	1	1	1	0	0,S+1	P	
			5	1	1	1	0	0,S+2	PCL	1
			6	1	1	1	0	0,S+3	PCH	1
		(7)	7	1	1	1	0	0,S+4	PBR	1
			1	1	1	1	1	PBR,PC	New Op Code	1
21d	Stack (Return from Subroutine) s (RTS) (1 Op Code) (1 byte) (6 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	0	PBR,PC+1	IO	1
			3	1	1	0	0	PBR,PC+1	IO	1
			4	1	1	1	0	0,S+1	PCL	1
			5	1	1	1	0	0,S+2	PCH	1
			6	1	1	0	0	0,S+2	IO	1
			1	1	1	1	1	PBR,PC	Op Code	1
★21e	Stack (Return from Subroutine Long) s (RTL) (1 Op Code) (1 byte) (6 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	0	PBR,PC+1	IO	1
			3	1	1	0	0	PBR,PC+1	IO	1
			4	1	1	1	0	0,S+1	NEW PCL	1
			5	1	1	1	0	0,S+2	NEW PCH	1
			6	1	1	1	0	0,S+3	NEW PBR	1
			1	1	1	1	1	NEW PBR,PC	Next Op Code	1
21f	Stack (Push) s (PHP PHA PHY PHX PHD PHK PHB) (7 Op Codes) (1 byte) (3 and 4 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	0	PBR,PC+1	IO	1
		(1)	3a	1	1	1	0	0,S	Register High	1
			3	1	1	1	0	0,S-1	Register Low	1
21g	Stack (Pull) s (PLP PLA PLY PLX PLD PLB) (Different than N6502) (6 Op Codes) (1 byte) (4 and 5 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	0	PBR,PC+1	IO	1
			3	1	1	0	0	PBR,PC+1	IO	1
			4	1	1	1	0	0,S+1	Register Low	1
		(1)	4a	1	1	1	0	0,S+2	Register High	1
★21h	Stack (Push Effective Indirect Address) s (PEI) (1 Op Code) (2 bytes) (6 and 7 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	DO	1
		(2)	2a	1	1	0	0	PBR,PC+1	IO	1
			3	1	1	1	0	0,D+DO	AAL	1
			4	1	1	1	0	0,D+DO+1	AAH	1
			5	1	1	1	0	0,S	AAH	0
			6	1	1	1	0	0,S-1	AAL	0
★21i	Stack (Push Effective Absolute Address) s (PEA) (1 Op Code) (3 bytes) (5 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	AAL	1
			3	1	1	0	1	PBR,PC+2	AAH	1
			4	1	1	1	0	0,S	AAH	0
			5	1	1	1	0	0,S-1	AAL	0
★21j	Stack (Push Effective Program Counter Relative Address) s (PER) (1 Op Code) (3 bytes) (6 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	Offset Low	1
			3	1	1	0	1	PBR,PC+2	Offset High	1
			4	1	1	0	0	PBR,PC+2	IO	1
			5	1	1	1	0	0,S	PCH+OFF+ CARRY	0
			6	1	1	1	0	0,S-1	PCL+OFFSET	0
★22	Stack Relative d,s (ORA,AND,EOR ADC STA LDA,CMP SBC) (8 Op Codes) (2 bytes) (4 and 5 cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	SO	1
			3	1	1	0	0	PBR,PC+1	IO	1
			4	1	1	1	0	0,S+SO	Data Low	1/0
		(1)	4a	1	1	1	0	0,S+SO+1	Data High	1/0
★23	Stack Relative Indirect Indexed (d,s),y (ORA AND EOR ADC STA LDA CMP SBC) (8 Op Codes) (2 bytes) (7 and 8 Cycles)		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	SO	1
			3	1	1	0	0	PBR,PC+1	IO	
			4	1	1	1	0	0,S+SO	AAL	1
			5	1	1	1	0	0,S+SO+1	AAH	1
			6	1	1	0	0	0,S+SO+1	IO	1
			7	1	1	1	0	DBR,AA+Y	Data Low	1/0
		(1)	7a	1		1	0	DBR,AA+Y+1	Data High	1/0
★24a	Block Move Positive (forward) xyc (MVP) (1 Op Code) (3 bytes) (7 cycles) x = Source Address y = Destination c = Number of Bytes to Move -1 x y Decrement MVP is used when the destination start address is higher (more positive) than the source start address  FFFFFF Dest. Start Source Start Dest. End Source End 000000		1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	DBA	1
			3	1	1	0	1	PBR,PC+2	SBA	1
		N-2	4	1	1	1	0	SBA,X	Source Data	1
		Byte	5	1	1	1	0	DBA,Y	Dest. Data	0
		C=2	6	1	1	0	0	DBA,Y	IO	1
			7	1	1	0	0	DBA,Y	IO	1
			1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	DBA	1
			3	1	1	0	1	PBR,PC+2	SBA	1
		N-1	4	1	1	1	0	SBA,X-1	Source Data	1
		Byte	5	1	1	1	0	DBA,Y-1	Dest. Data	0
		C=1	6	1	1	0	0	DBA,Y-1	IO	1
			7	1	1	0	0	DBA,Y-1	IO	1
			1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	DBA	1
		N Byte	3	1	1	0	1	PBR,PC+2	SBA	1
		Last	4	1	1	1	0	SBA,X-2	Source Data	1
		C=0	5	1	1	1	0	DBA,Y-2	Dest. Data	0
			6	1	1	0	0	DBA,Y-2	IO	1
			7	1	1	0	0	DBA,Y-2	IO	1
			1	1	1	1	1	PBR,PC+3	Next Op Code	1
★24b	Block Move Negative (backward) xyc (MVN) (1 Op Code) (3 bytes) (7 cycles) x = Source Address y = Destination c = Number of Bytes to Move -1 x y Increment  FFFFFF Source End Dest. End Source Start Dest. Start 000000 MVN is used when the destination start address is lower (more negative) than the source start address		1	1	1		1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	DBA	1
		N-2	3	1	1	0	1	PBR,PC+2	SBA	1
		Byte	4	1	1	1	0	SBA,X	Source Data	1
		C=2	5	1	1	1	0	DBA,Y	Dest. Data	0
			6	1	1	0	0	DBA,Y	IO	
			7	1		0	0	DBA,Y	IO	1
			1	1	1	1	1	PBR,PC	Op Code	1
			2		1	0	1	PBR,PC+1	DBA	1
		N-1	3	1	1	0	1	PBR,PC+2	SBA	1
		Byte	4	1	1	1	0	SBA,X+1	Source Data	1
		C=1	5	1	1	1	0	DBA,Y+1	Dest. Data	0
			6	1	1	0	0	DBA,Y+1	IO	1
			7	1	1	0	0	DBA,Y+1	IO	1
			1	1	1	1	1	PBR,PC	Op Code	1
			2	1	1	0	1	PBR,PC+1	DBA	1
		N Byte	3	1	1	0	1	PBR,PC+2	SBA	1
		C=0	4	1	1	1	0	SBA,X+2	Source Data	1
			5	1	1	1	0	DBA,Y+2	Dest. Data	0
			6	1	1	0	0	DBA,Y+2	IO	1
			7	1	1	0	0	DBA,Y+2	IO	1
			1	1	1	1	1	PBR,PC+3	Next Op Code	1

Notes

(1) Add 1 byte (for immediate only) for M=0 or X=0 (i.e. 16 bit data), add 1 cycle for M=0 or X=0.

(2) Add 1 cycle for direct register low (DL) not equal 0.

(3) Special case for aborting instruction. This is the last cycle which may be aborted or the Status, PBR or DBR registers will be updated.

(4) Add 1 cycle for indexing across page boundaries, or write, or X=0. When X=1 or in the emulation mode, this cycle contains invalid addresses.

(5) Add 1 cycle if branch is taken.

(6) Add 1 cycle if branch is taken across page boundaries in 6502 emulation mode (E=1).

(7) Subtract 1 cycle for 6502 emulation mode (E=1).

(8) Add 1 cycle for REP, SEP.

(9) Wait at cycle 2 for 2 cycles after $\overline{NMI}$ or $\overline{IRQ}$ active input.

(10) R/$\overline{W}$ remains high during Reset.

Abbreviations

AAB	Absolute Address Bank
AAH	Absolute Address High
AAL	Absolute Address Low
AAVH	Absolute Address Vector High
AAVL	Absolute Address Vector Low
C	Accumulator
D	Direct Register
DBA	Destination Bank Address
DBR	Data Bank Register
DO	Direct Offset
IDH	Immediate Data High
IDL	Immediate Data Low
IO	Internal Operation
P	Status Register
PBR	Program Bank Register
PC	Program Counter
R-M-W	Read Modify Write
S	Stack Address
SBA	Source Bank Address
SO	Stack Offset
VA	Vector Address
x, y	Index Registers
★	New W65C816/802 Addressing Modes
●	New W65C02 Addressing Modes
Blank	NMOS 6502 Addressing Modes

Recommended W65C816 and W65C802 Assembler Syntax Standards

Directives

Assembler directives are those parts of the assembly language source program which give directions to the assembler; this includes the definition of data area and constants within a program. This standard excludes any definitions of assembler directives

Comments

An assembler should provide a way to use any line of the source program as a comment. The recommended way of doing this is to treat any blank line, or any line that starts with a semi-colon or an asterisk as a comment Other special characters may be used as well

The Source Line

Any line which causes the generation of a single W65C816 or W65C802 machine language instruction should be divided into four fields: a label field, the operation code, the operand, and the comment field

The Label Field—The label field begins in column one of the line. A label must start with an alphabetic character, and may be followed by zero or more alphanumeric characters. An assembler may define an upper limit on the number of characters that can be in a label, so long as that upper limit is greater than or equal to six characters. An assembler may limit the alphabetic characters to upper-case characters if desired. If lower-case characters are allowed, they should be treated as identical to their upper-case equivalents. Other characters may be allowed in the label, so long as their use does not conflict with the coding of operand fields

The Operation Code Field—The operation code shall consist of a three character sequence (mnemonic) from Table 3. It shall start no sooner than column 2 of the line, or one space after the label if a label is coded

Many of the operation codes in Table 3 have duplicate mnemonics, when two or more machine language instructions have the same mnemonic, the assembler resolves the difference based on the operand.

If an assembler allows lower-case letters in labels, it must also allow lower-case letters in the mnemonic. When lower-case letters are used in the mnemonic, they shall be treated as equivalent to the upper-case counterpart. Thus, the mnemonics LDA, lda, and LdA must all be recognized, and are equivalent

In addition to the mnemonics shown in Table 3, an assembler may provide the alternate mnemonics shown in Table 6

Table 7. Alternate Mnemonics

Standard	Alias
BCC	BLT
BCS	BGE
CMP A	CMA
DEC A	DEA
INC A	INA
JSL	JSR
JML	JMP
TCD	TAD
TCS	TAS
TDC	TDA
TSC	TSA
XBA	SWA

JSL should be recognized as equivalent to JSR when it is specified with a long absolute address. JML is equivalent to JMP with long addressing forced

The Operand Field—The operand field may start no sooner than one space after the operation code field. The assembler must be capable of at least twenty-four bit address calculations. The assembler should be capable of specifying addresses as labels, integer constants, and hexadecimal constants. The assembler must allow addition and subtraction in the operand field. Labels shall be recognized by the fact that they start alphabetic characters. Decimal numbers shall be recognized as containing only the decimal digits 0...9. Hexadecimal constants shall be recognized by prefixing the constant with a "$" character, followed by zero or more of either the decimal digits or the hexadecimal digits "A"... "F". If lower-case letters are allowed in the label field, then they shall also be allowed as hexadecimal digits

All constants, no matter what their format, shall provide at least enough precision to specify all values that can be represented by a twenty-four bit signed or unsigned integer represented in two's complement notation.

Table 8 shows the operand formats which shall be recognized by the assembler. The symbol **d** is a label or value which the assembler can recognize as being less than $100. The symbol **a** is a label or value which the assembler can recognize as greater the $FF but less than $10000; the symbol **al** is a label or value that the assembler can recognize as being greater than $FFFF. The symbol EXT is a label which cannot be located by the assembler at the time the instruction is assembled Unless instructed otherwise, an assembler shall assume that EXT labels are two bytes long. The symbols **r** and **rl** are 8 and 16 bit signed displacements calculated by the assembler.

Note that the operand does not determine whether or not immediate addressing loads one or two bytes, this is determined by the setting of the status register. This forces the requirement for a directive or directives that tell the assembler to generate one or two bytes of space for immediate loads. The directives provided shall allow separate settings for the accumulator and index registers.

The assembler shall use the <, >, and ^ characters after the # character in immediate address to specify which byte or bytes will be selected from the value of the operand. Any calculations in the operand must be performed before the byte selection takes place Table 7 defines the action taken by each operand by showing the effect of the operator on an address. The column that shows a two byte immediate value show the bytes in the order in which they appear in memory The coding of the operand is for an assembler which uses 32 bit address calculations, showing the way that the address should be reduced to a 24 bit value

Table 8. Byte Selection Operator

Operand	One Byte Result	Two Byte Result	
#$01020304	04	04	03
#<$01020304	04	04	03
#>$01020304	03	03	02
# ^$01020304	02	02	01

In any location in an operand where an address, or expression resulting in an address, can be coded, the assembler shall recognize the prefix characters <, |, and >, which force one byte (direct page), two byte (absolute) or three byte (long absolute) addressing. In cases where the addressing mode is not forced, the assembler shall assume that the address is two bytes unless the assembler is able to determine the type of addressing required by context, in which case that addressing mode will be used Addresses shall be truncated without error if an addressing mode is forced which does not require the entire value of the address. For example,

```
LDA   $0203          LDA   |$010203
```

are completely equivalent. If the addressing mode is not forced, and the type of addressing cannot be determined from context, the assembler shall assume that a two byte address is to be used. If an instruction does not have a short addressing mode (as in LDA, which has no direct page indexed by Y) and a short address is used in the operand, the assembler shall automatically extend the address by padding the most significant bytes with zeroes in order to extend the address to the length needed. As with immediate addressing, any expression evaluation shall take place before the address is selected, thus, the address selection character is only used once, before the address of expression.

The ! (exclamation point) character should be supported as an alternative to the | (vertical bar).

A long indirect address is indicated in the operand field of an instruction by surrounding the direct page address where the indirect address is found by square brackets; direct page addresses which contain sixteen-bit addresses are indicated by being surrounded by parentheses

The operands of a block move instruction are specified as source bank, destination bank—the opposite order of the object bytes generated.

Comment Field—The comment field may start no sooner than one space after the operation code field or operand field depending on instruction type

Table 9. Address Mode Formats

Addressing Mode	Format	
Immediate	#d	
	#a	
	#al	
	#EXT	
	#<d	
	#<a	
	#<al	
	#<EXT	
	#>d	
	#>a	
	#>al	
	#>EXT	
	#^d	
	#^a	
	#^al	
	#^EXT	
Absolute	!d	
	!a	
	a	
	!al	
	!EXT	
	EXT	
Absolute Long	>d	
	>a	
	>al	
	al	
	>EXT	
Direct Page	d	
	<d	
	<a	
	<al	
	<EXT	
Accumulator	A	
Implied Addressing	(no operand)	
Direct Indirect Indexed	(d),y	
	(<d),y	
	(<a),y	
	(<al),y	
	(<EXT),y	
Direct Indirect Indexed Long	[d],y	
	[<d],y	
	[<a],y	
	[<al],y	
	[<EXT],y	
Direct Indexed Indirect	(d,x)	
	(<d,x)	
	(<a,x)	
	(<al,x)	
	(<EXT,x)	
Direct Indexed by X	d,x	
	<d,x	
	<a,x	
	<al,x	
	<EXT,x	
Direct Indexed by Y	d,y	
	<d,y	
	<a,y	
	<al,y	
	<EXT,y	
Absolute Indexed by X	d,x	
	!d,x	
	a,x	
	!a,x	
	!al,x	
	!EXT,x	
	EXT,x	
Absolute Indexed by Y	!d y	
	d,y	
	a,y	
	!a,y	
	!al y	
	!EXT,y	
	EXT,y	
Absolute Long Indexed by X	>d,x	
	>a,x	
	>al,x	
	al,x	
	>EXT,x	
Program Counter Relative and Program Counter Relative Long	d	(the assembler calculates r and rl)
	a	
	al	
	EXT	
Absolute Indirect	(d)	
	(!d)	
	(a)	
	(!a)	
	(!al)	
	(EXT)	
Direct Indirect	(d)	
	(<a)	
	(<al)	
	(<EXT)	
Direct Indirect Long	[d]	
	[<a]	
	[<al]	
	[<EXT]	
Absolute Indexed	(d,x)	
	(!d,x)	
	(a,x)	
	(!a,x)	
	(!al,x)	
	(EXT,x)	
	(!EXT,x)	
Stack Addressing	(no operand)	
Stack Relative Indirect Indexed	(d,s),y	
	(<d,s),y	
	(<a,s),y	
	(<al,s),y	
	(<EXT,s),y	
Block Move	d,d	
	d,a	
	d,al	
	d,EXT	
	a,d	
	a,a	
	a,al	
	a,EXT	
	al,d	
	al,a	
	al,al	
	al,EXT	
	EXT,d	
	EXT,a	
	EXT,al	
	EXT,EXT	

Note: The alternate ! (exclamation point) is used in place of the | (vertical bar).

Table 10. Addressing Mode Summary

	Instruction Times In Memory Cycles		Memory Utilization In Number of Program Sequence Bytes	
Address Mode	Original 8 Bit NMOS 6502	New W65C816	Original 8 Bit NMOS 6502	New W65C816
1. Immediate	2	$2^{(3)}$	2	$2^{(3)}$
2 Absolute	$4^{(5)}$	$4^{(3,5)}$	3	3
3. Absolute Long	—	$5^{(3)}$	—	4
4. Direct	$3^{(5)}$	$3^{(3\ 4\ 5)}$	2	2
5 Accumulator	2	2	1	1
6. Implied	2	2	1	1
7. Direct Indirect Indexed (d),y	$5^{(1)}$	$5^{(1\ 3\ 4)}$	2	2
8. Direct Indirect Indexed Long [d], y	—	$6^{(3\ 4)}$	—	2
9 Direct Indexed Indirect (d,x)	6	$6^{(3,4)}$	2	2
10. Direct, X	$4^{(5,}$	$4^{(3\ 4\ 5)}$	2	2
11 Direct, Y	4	$4^{(3\ 4)}$	2	2
12. Absolute, X	$4^{(1\ 5,}$	$4^{(1,3,5)}$	3	3
13. Absolute Long, X	—	$5^{(3)}$	—	4
14 Absolute, Y	$4^{(1)}$	$4^{(1\ 3)}$	3	3
15. Relative	$2^{(1\ 2)}$	$2^{(2)}$	2	2
16. Relative Long	—	$3^{(2)}$	—	3
17 Absolute Indirect (Jump)	5	5	3	3
18 Direct Indirect	–	$5^{(3\ 4)}$	—	2
19. Direct Indirect Long		$6^{(3\ 4)}$	—	2
20. Absolute Indexed Indirect (Jump)	–	6	–	3
21 Stack	3-7	3-8	1-3	1-4
22 Stack Relative	—	$4^{(3)}$	—	2
23. Stack Relative Indirect Indexed		$7^{(3)}$	—	2
24. Block Move X, Y, C (Source, Destination, Block Length)	-	7	–	3

NOTES

1. Page boundary, add 1 cycle if page boundary is crossed when forming address
2. Branch taken, add 1 cycle if branch is taken
3. M = 0 or X = 0, 16 bit operation, add 1 cycle, add 1 byte for immediate
4. Direct register low (DL),not equal zero, add 1 cycle
5 Read-Modify-Write, add 2 cycles for M = 1, add 3 cycles for M = 0

Caveats and Application Information

Stack Addressing

When in the Native mode, the Stack may use memory locations 000000 to 00FFFFF. The effective address of Stack, Stack Relative, and Stack Relative Indirect Indexed addressing modes will always be within this range. In the Emulation mode, the Stack address range is 000100 to 0001FF. The following opcodes and addressing modes will increment or decrement beyond this range when accessing two or three bytes

JSL, JSR(a,x), PEA, PEI; PER; PHD, PLD, RTL, d,s; (d,s),y

Direct Addressing

The Direct Addressing modes are often used to access memory registers and pointers. The effective address generated by Direct; Direct,X and Direct,Y addressing modes will always be in the Native mode range 000000 to 00FFFF. When in the Emulation mode, the direct addressing range is 000000 to 0000FF, except for [Direct] and [Direct],Y addressing modes and the PEI instruction which will increment from 0000FE or 0000FF into the Stack area

When in the Emulation mode and DH is not equal to zero, the direct addressing range is 00DH00 to 00DHFF, except for [Direct] and [Direct],Y addressing modes and the PEI instruction which will increment from 00DHFE or 00DHFF into the next higher page

When in the Emulation mode and DL in not equal to zero, the direct addressing range is 000000 to 00FFFF

Absolute Indexed Addressing (W65C816 Only)

The Absolute Indexed addressing modes are used to address data outside the direct addressing range. The W65C02 and W65C802 addressing range is 0000 to FFFF. Indexing from page FFXX may result in a 00YY data fetch when using the W65C02 or W65C802 In contrast, indexing from page ZZFFXX may result in ZZ+1,00YY when using the W65C816

Future Microprocessors (i.e., W65C832)

Future WDC microprocessors will support all current W65C816 operating modes for both index and offset address generation

ABORT Input (W65C816 Only)

ABORT should be held low for a period not to exceed one cycle Also, if ABORT is held,low during the Abort Interrupt sequence, the Abort Interrupt will be aborted. It is not recommended to abort the Abort Interrupt. The ABORT internal latch is cleared during the second cycle of the Abort Interrupt. Asserting the ABORT input after the following instruction cycles will cause registers to be modified

- **Read-Modify-Write:** Processor status modified if ABORT is asserted after a modify cycle.
- **RTI:** Processor status will be modified if ABORT is asserted after cycle 3.
- **IRQ, NMI, ABORT BRK, COP:** When ABORT is asserted after cycle 2, PBR and DBR will become 00 (Emulation mode) or PBR will become 00 (Native mode).

The Abort Interrupt has been designed for virtual memory systems. For this reason, asynchronous ABORT's may cause undesirable results due to the above conditions

VDA and VPA Valid Memory Address Output Signals (W65C816 Only)

When VDA or VPA are high and during all write cyclés, the Address Bus is always valid. VDA and VPA should be used to qualify all memory cycles. Note that when VDA and VPA are both low, invalid addresses may be generated. The Page and Bank addresses could also be invalid. This will be due to low byte addition only. The cycle when only low byte addition occurs is an optional cycle for instructions which read memory when the Index Register consists of 8 bits. This optional cycle becomes a standard cycle for the Store instruction, all instructions using the 16-bit Index Register mode, and the Read-Modify-Write instruction when using 8- or 16-bit Index Register modes

Apple II, IIe, IIc and II+ Disk Systems (W65C816 Only)

VDA and VPA should not be used to qualify addresses during disk operation on Apple systems. Consult your Apple representative for hardware/software configurations.

DB/BA Operation when RDY is Pulled Low (W65C816 Only)

When RDY is low, the Data Bus is held in the data transfer state (i.e , $\phi 2$ high) The Bank address external transparent latch should be latched when the $\phi 2$ clock or RDY is low

M/X Output (W65C816 Only)

The M/X output reflects the value of the M and X bits of the processor Status Register The REP, SEP and PLP instructions may change the state of the M and X bits. Note that the M/X output is invalid during the instruction cycle following REP, SEP and PLP instruction execution This cycle is used as the opcode fetch cycle of the next instruction

All Opcodes Function in All Modes of Operation

It should be noted that all opcodes function in all modes of operation However, some instructions and addressing modes are intended for W65C816 24-bit addressing and are therefore less useful for the W65C802 The following is a list of instructions and addressing modes which are primarily intended for W65C816 use

JSL; RTL, [d]; [d],y, JMP al, JML, al, al,x

The following instructions may be used with the W65C802 even though a Bank Address is not multiplexed on the Data Bus

PHK; PHB; PLB

The following instructions have "limited" use in the Emulation mode.

- The REP and SEP instructions cannot modify the M and X bits when in the Emulation mode In this mode the M and X bits will always be high (logic 1).
- When in the Emulation mode, the MVP and MVN instructions use the X and Y Index Registers for the memory address Also, the MVP and MVN instructions can only move data within the memory range 0000 (Source Bank) to 00FF (Destination Bank) for the W65C816, and 0000 to 00FF for the W65C802

Indirect Jumps

The JMP (a) and JML (a) instructions use the direct Bank for indirect addressing, while JMP (a,x) and JSR (a,x) use the Program Bank for indirect address tables

Switching Modes

When switching from the Native mode to the Emulation mode, the X and M bits of the Status Register are set high (logic 1), the high byte of the Stack is set to 01, and the high bytes of the X and Y Index Registers are set to 00. To save previous values, these bytes must always be stored before changing modes. Note that the low byte of the S, X and Y Registers and the low and high byte of the Accumulator (A and B) are not affected by a mode change

How Hardware Interrupts, BRK, and COP Instructions Affect the Program Bank and the Data Bank Registers

When in the Native mode, the Program Bank register (PBR) is cleared to 00 when a hardware interrupt, BRK or COP is executed In the Native mode, previous PBR contents is automatically saved on Stack

In the Emulation mode, the PBR and DBR registers are cleared to 00 when a hardware interrupt, BRK or COP is executed. In this case, previous contents of the PBR are not automatically saved

Note that a Return from Interrupt (RTI) should always be executed from the same "mode" which originally generated the interrupt

Binary Mode

The Binary mode is set whenever a hardware or software interrupt is executed. The D flag within the Status Register is cleared to zero.

WAI Instruction

The WAI instruction pulls RDY low and places the processor in the WAI "low power" mode. NMI, IRQ or RESET will terminate the WAI condition and transfer control to the interrupt handler routine. Note that an ABORT input will abort the WAI instruction, but will not restart the processor. When the Status Register I flag is set (IRQ disabled), the IRQ interrupt will cause the next instruction (following the WAI instruction) to be executed without going to the IRQ interrupt handler. This method results in the highest speed response to an IRQ input. When an interrupt is received after an $\overline{\text{ABORT}}$ which occurs during the WAI instruction the processor will return to the WAI instruction Other than $\overline{\text{RES}}$ (highest priority), $\overline{\text{ABORT}}$ is the next highest priority followed by $\overline{\text{NMI}}$ or $\overline{\text{IRQ}}$ interrupts

STP Instruction

The STP instruction disables the $\phi 2$ clock to all circuitry When disabled, the $\phi 2$ clock is held in the high state In this case, the Data Bus will remain in the data transfer state and the Bank address will not be multiplexed onto the Data Bus. Upon executing the STP instruction, the $\overline{\text{RES}}$ signal is the only input which can restart the processor. The processor is restarted by enabling the $\phi 2$ clock, which occurs on the falling edge of the $\overline{\text{RES}}$ input Note that the external oscillator must be stable and operating properly before $\overline{\text{RES}}$ goes high

COP Signatures

Signatures 00-7F may be user defined, while signatures 80-FF are reserved for instructions on future microprocessors (i e., W65C832). Contact WDC for software emulation of future microprocessor hardware functions.

WDM Opcode Use

The WDM opcode will be used on future microprocessors For example, the new W65C832 uses this opcode to provide 32-bit floating-point and other 32-bit math and data operations Note that the W65C832 will be a plug-to-plug replacement for the W65C816, and can be used where high-speed, 32-bit math processing is required The W65C832 will be available in the near future

RDY Pulled During Write

The NMOS 6502 does not stop during a write operation In contrast, both the W65C02 and the W65C816 do stop during write operations The W65C802 stops during a write when in the Native mode, but does not stop when in the Emulation mode

MVN and MVP Affects on the Data Bank Register

The MVN and MVP instructions change the Data Bank Register to the value of the second byte of the instruction (destination bank address)

Interrupt Priorities

The following interrupt priorities will be in effect should more than one interrupt occur at the same time.

$\overline{\text{RES}}$	Highest Priority
$\overline{\text{ABORT}}$	
$\overline{\text{NMI}}$	
$\overline{\text{IRQ}}$	Lowest Priority

Transfers from 8-Bit to 16-Bit, or 16-Bit to 8-Bit Registers

All transfers from one register to another will result in a full 16-bit output from the source register. The destination register size will determine the number of bits actually stored in the destination register and the values stored in the processor Status Register. The following are always 16-bit transfers, regardless of the accumulator size:

TCS, TSC; TCD; TDC

Stack Transfers

When in the Emulation mode, a 01 is forced into SH In this case, the B Accumulator will not be loaded into SH during a TCS instruction When in the Native mode, the B Accumulator is transferred to SH Note that in both the Emulation and Native modes, the full 16 bits of the Stack Register are transferred to the A, B and C Accumulators, regardless of the state of the M bit in the Status Register

WDC Toolbox System-Emulator

Features

- Real-Time emulation of the W65C802/816 and the W65C02
- Uses an inexpensive Apple IIe Computer as host (software provided)
- 18K bytes of Emulation RAM, mappable in 2K blocks
- Optional RAM expansion to 256K
- Optional hardware Real-Time Trace Board
- Optional 802/816 Emulation Pod Unit
- Single-Step
- 48 bit trace memory of up to 2048 machine cycles
- Three 40-bit breakpoint control registers providing
  —Break on Address
  —Break on Data
  —Break on Control
  —Break on User Status
  —Break on Nth Occurance
  —Coast Mode
- Microsecond execution timer
- Also available in In-Circuit-Evaluation chip or system test configuration

Product Overview

The Toolbox System-Emulator consists of a Main Unit and Interface Card that plugs into one of the Apple Computer's expansion slots. The Main Unit provides all necessary logic for breakpointing, single-stepping and mapping In this configuration the user may perform basic debug operations or use the Toolbox in the Evaluation Mode

With the optional Real-Time Trace Board, the user now has 40 bits of trace memory within a window of 2048 machine cycles. A optional Emulation RAM Expansion Board is also available which increases the user's emulation RAM by 64K bytes or 256K bytes, with memory configuration under software control

The Toolbox may be used with or without the optional Pod Unit. With the Pod Unit, the user can plug into the prototype microprocessor socket for hardware debug Since the Main Unit remains the same regardless of the microprocessor used, the user does not have to learn a new set of Toolbox commands for each type of processor

Apple IIe is a trademark of Apple Computer, Inc

Additional Information

For additional information on the W65C802/816, refer to the following publications

Programming the 65816
William Labiak
SYBEX, Inc.
2344 Sixth St.
Berkeley, CA 94710

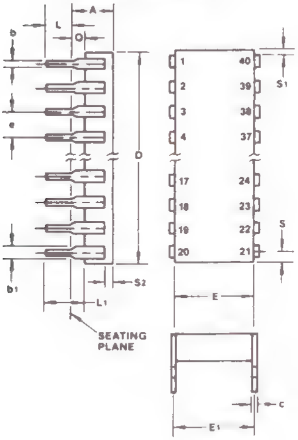
The 6502, 65C02 and 65816 Handbook
Steve Hendrix
Weber Systems, Inc.
8437 Mayfield Rd
Chesterland, OH 44026

65816/65802 Assembly Language Programming
Michael Fisher
Osborne McGraw-Hill
2600 Tenth St.
Berkeley, CA 94710

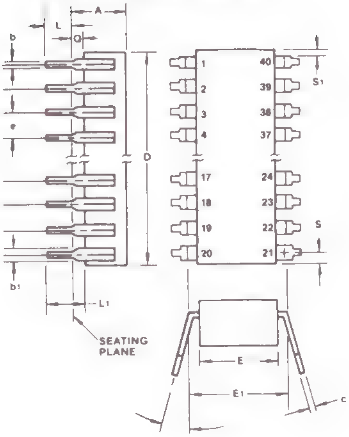
Programming the 65816 Including the 6502, 65C02, and 65802
David Eyes and Ron Lichty
Prentice Hall Press
A Division of Simon & Schuster, Inc.
Gulf & Western Bldg
One Gulf & Western Plaza
New York, NY 10023

Packaging Information

Ceramic Package

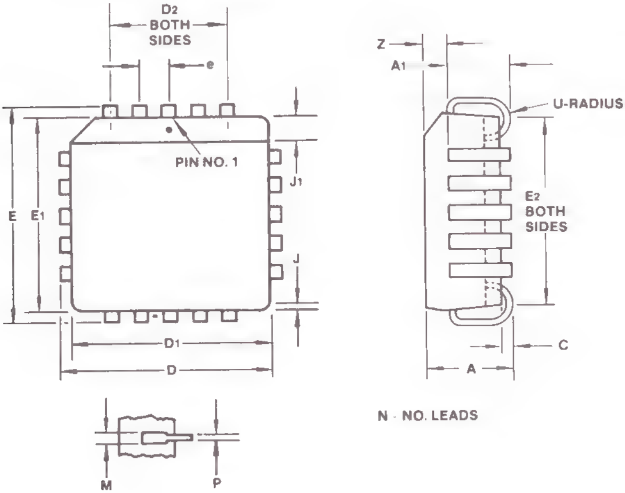


Plastic & Cerdip Package



40 PIN PACKAGE				
SYM BOL	INCHES		MILL METERS	
	MIN	MAX	MIN	MAX
A		0.225		5.72
b	0.014	0.023	0.36	0.58
b1	0.030	0.070	0.76	1.78
c	0.008	0.015	0.20	0.38
D		2.096		53.24
E	0.510	0.620	12.95	15.75
E1	0.520	0.630	13.21	16.00
e	0.100 BSC		2.54 BSC	
L	0.125	0.200	3.18	5.08
L1	0.150		3.81	
Q	0.020	0.060	0.51	1.52
S		0.098		2.49
S1	0.005		0.13	
S2	0.005		0.13	
α	0°	15°	0°	15°

Plastic Leaded Chip Carrier



44-LEAD CARRIER				
SYM BOL	INCHES		MILLIMETERS	
	MIN	MAX	MIN	MAX
A	0.165	0.180	4.20	4.57
A1	0.090	0.120	2.29	3.04
C	0.020		0.51	
D	0.685	0.695	17.40	17.65
D1	0.650	0.656	16.510	16.662
D2	0.500 REF		12.70 BSC	
E	0.685	0.695	17.40	17.65
E1	0.650	0.656	16.510	16.662
E2	0.590	0.630	14.99	16.00
e	0.050 TYP		1.27 TYP	
J	–	0.020	–	0.51
J1	0.042	0.048	1.067	1.219
M	0.026	0.032	0.661	0.812
N	44		44	
P	0.013	0.021	0.331	0.533
Z	0.042	0.056	1.07	1.42

NOTES:

1. Power supply pins not available on the 40-pin version. These power supply pins have been added for improved high performance.
2. New pins, not available on 40-pin version

Ordering Information

W 65C816 PL I -2

Description

WC—Custom
W—Standard

Product Identification Number

Package

P—Plastic	E—Leadless Chip Carrier
C—Ceramic	X—Dice
D—Cerdip	PL—Plastic Chip Carrier

Temperature/Processing

Blank —0° C to 70° C
I — -40° C to +85° C
M— 55° C to +125° C

Performance Designator

Designators selected for speed and power specifications

Blank	—2 MHz
-4	4 MHz
-6	6 MHz
-8	8 MHz

Sales Offices:

Technical or sales assistance may be requested from

The Western Design Center, Inc
2166 East Brown Road
Mesa, Arizona 85203
602/962-4545
TLX 6835057

WARNING:

MOS CIRCUITS ARE SUBJECT TO DAMAGE FROM STATIC DISCHARGE

Internal static discharge circuits are provided to minimize part damage due to environmental static electrical charge build ups Industry established recommendations for handling MOS circuits include

1 Ship and store product in conductive shipping tubes or in conductive foam plastic Never ship or store product in non conductive plastic containers or non conductive plastic foam material
2 Handle MOS parts only at conductive work stations
3 Ground all assembly and repair tools

Represented in your area by:

WDC reserves the right to make changes at any time and without notice

Information contained herein is provided gratuitously and without liability, to any user Reasonable efforts have been made to verify the accuracy of the information but no guarantee whatsoever is given as to the accuracy or as to its applicability to particular uses In every instance it must be the responsibility of the user to determine the suitability of the products for each application WDC products are not authorized for use as critical components in life support devices or systems Nothing contained herein shall be construed as a recommendation to use any product in violation of existing patents or other rights of third parties The sale of any WDC product is subject to all WDC Terms and Conditions of Sale and Sales Policies, copies of which are available upon request

The Western Design Center, Inc. 2166 E. Brown Rd./Mesa, AZ 85203 FAX6028356442
602/962-4545/TLX 6835057
Revised August 1986
Published in U.S.A. November 1986

APPENDIX 3

The Apple IIGS Tool Sets

Approximately half of the Apple IIGS tool sets are located in ROM. (They are marked with asterisks in the table below.) When ProDOS 16 starts up, it installs RAM-based bug fixes and enhancements to the ROM-based tool sets by executing the TOOL.SETUP program in the SYSTEM/SYSTEM.SETUP/ subdirectory of the boot volume.

RAM-based tool sets are stored on disk in files with names of the form TOOLxxx (where xxx represents the three-digit tool set number in decimal form) in the SYSTEM/TOOLS/ subdirectory of the boot volume. You can install these tool sets from inside an application with the Tool Locator's LoadTools and LoadOneTool functions (see chapter 3).

The macro files referred to in the following table contain the macro definitions for tool set functions in assembly language form. These files are included with the Apple IIGS Programmer's Workshop (APW).

Table of Apple IIGS Tool Sets

Tool Set Number	*Tool Set Name*	*APW Macro File*
001	*Tool Locator	M16.LOCATOR
002	*Memory Manager	M16.MEMORY
003	*Miscellaneous Tool Set	M16.MISCTOOL
004	*QuickDraw II	M16.QUICKDRAW
005	*Desk Manager	M16.DESK
006	*Event Manager	M16.EVENT
007	*Scheduler	M16.SCHEDULER

Tool Set Number	*Tool Set Name*	*APW Macro File*
008	*Sound Manager	M16.SOUND
009	*DeskTop Bus Tool Set	M16.ADB
010	*Floating-Point Numerics (SANE)	M16.SANE
011	*Integer Math Tool Set	M16.INTMATH
012	*Text Tool Set	M16.TEXTTOOL
013	*RAM Disk Tool Set	[internal use]
014	Window Manager	M16.WINDOW
015	Menu Manager	M16.MENU
016	Control Manager	M16.CONTROL
017	System Loader	M16.LOADER
018	QuickDraw Auxiliary Tool Set	M16.QDAUX
019	Print Manager	M16.PRINT
020	LineEdit	M16.LINEEDIT
021	Dialog Manager	M16.DIALOG
022	Scrap Manageı	M16.SCRAP
023	Standard File Operations Tool Set	M16.STDFILE
024	Disk Utilities	[none]
025	Note Synthesizer	M16.NOTESYN
026	Note Sequencer	[none]
027	Font Manager	M16.FONT
028	List Manager	M16.LIST

* = tool set in ROM

APPENDIX 4

Number-Conversion Functions

The Integer Math Tool Set (tool set 11) includes nine functions you can use to convert numeric data from one numbering system to another. (See the table below.) The three numbering systems supported are binary, decimal, and hexadecimal. The purpose of this appendix is to explain how to use these conversion functions in your programs.

Table of Integer Math Tool Set Number-Conversion Functions

Function	*Description*
Int2Hex	Converts unsigned integer to hex ASCII string
Long2Hex	Converts unsigned long integer to hex ASCII string
HexIt	Converts unsigned integer to hex ASCII string on stack
Hex2Int	Converts hex ASCII string to unsigned integer
Hex2Long	Converts hex ASCII string to unsigned long integer
Int2Dec	Converts signed/unsigned integer to decimal ASCII string
Long2Dec	Converts signed/unsigned long integer to decimal ASCII string
Dec2Int	Converts signed/unsigned decimal ASCII string to integer
Dec2Long	Converts signed/unsigned decimal ASCII string to long integer

BINARY TO HEXADECIMAL

There are three functions for converting unsigned binary numbers to hexadecimal form: Int2Hex, Long2Hex, and HexIt.

Int2Hex converts a 16-bit unsigned binary integer to an ASCII string representing its value in hexadecimal form. Here is how to use it:

```
          PushWord TheNumber    ;Number to convert
          PushPtr HexString     ;Push pointer to hex string
          PushWord #4           ;Length of hex string
          _Int2Hex
          RTS

HexString DS    4               ;(standard ASCII; bit 7 = 0)
```

The hexadecimal string returned by Int2Hex is right-justified and padded on the left with ASCII 0 characters. In most situations, you will reserve four bytes for the string because the maximum value of a 16-bit integer is $FFFF. If you do not leave enough room, an error of $0B04 (string overflow) will be returned.

You can also use HexIt to convert a 16-bit unsigned integer to a 4-byte ASCII string of hexadecimal characters. The difference between it and Int2Hex is that the result is returned on the stack:

```
PHA                     ;Space for result (long)
PHA
PushWord TheNumber      ;Number to convert
_HexIt
PopLong HexResult       ;Pop the result
```

The first bytes popped from the stack are the low-order hexadecimal digits.

Long2Hex works much like Int2Hex. The key difference is that the number to be converted is a 32-bit unsigned long integer:

```
          PushLong TheNumber    ;Push number to convert
          PushPtr HexString     ;Push pointer to hex string
          PushWord #8           ;Length of hex string
          _Long2Hex
          RTS

HexString DS    8               ;(standard ASCII; bit 7 = 0)
```

Notice that eight bytes are reserved for the hexadecimal string result because the maximum value for a long integer is $FFFFFFFF.

HEXADECIMAL TO BINARY

To convert ASCII strings representing hexadecimal numbers to binary form, use Hex2Int (integers) or Hex2Long (long integers). In both cases, the binary result is returned on the stack. Here is how to call Hex2Int:

```
                    PHA                     ;space for result
                    PushPtr HexNumber       ;Pointer to hex number
                    PushWord #4             ;Length of hex number string
                    _Hex2Int
                    PopWord BinResult       ;pop the result

          HexNumber DC      C'7EAF'         ;Number to convert
```

The length parameter cannot be greater than 4 since an integer cannot exceed $FFFF.

Hex2Long converts hexadecimal strings to 32-bit numbers:

```
                    PHA                     ;space for result
                    PHA
                    PushPtr HexNumber       ;Pointer to hex number
                    PushWord #8             ;Length of hex string
                    _Hex2Long
                    PopLong BinResult       ;pop the result

          HexNumber DC      C'00E12000'     ;Number to convert
```

The length parameter cannot be greater than 8 for long integers.

BINARY TO DECIMAL

The binary-to-decimal conversion functions are useful for transforming binary numbers to the more-recognizable decimal form.

Int2Dec converts a 16-bit signed or unsigned binary number to an ASCII string representing the number in decimal form. If the number is negative, the string contains a minus sign to the left of the first digit in the string. No plus sign is inserted if the number is positive, however.

Here is the calling sequence for Int2Dec:

```
                    PushWord TheNumber      ;The number to convert
                    PushPtr DecString       ;Pointer to decimal result
                    PushWord #6             ;Size of string
                    PushWord SignedFlag     ;0 = unsigned, 1 = signed
                    _Int2Dec
                    RTS

          DecString DS      6               ;ASCII decimal string
```

The value of the SignedFlag parameter indicates whether the number is to be considered signed or unsigned. The size of the string should be at least 6 to accommodate the longest result. The result is right-justified and padded on the left with blanks. Each character of the result has the high-order bit cleared to 0.

Long2Dec works much as Int2Dec does:

```
              PushLong TheNumber     ;The number to convert
              PushPtr DecString      ;Pointer to decimal result
              PushWord #11           ;Size of string
              PushWord SignedFlag    ;0 = unsigned, 1 = signed
              _Int2Dec
              RTS

    DecString DS    11               ;ASCII decimal string
```

The key differences between this and Int2Dec are that the number to be converted is a long word and that the size of the string returned might be up to eleven characters long.

DECIMAL TO BINARY

Most programs ask users to enter numeric information in decimal form. To convert decimal numbers to the binary form the 65816 understands, use Dec2Int or Dec2Long.

Dec2Int converts the ASCII string for a decimal number to a 16-bit unsigned or signed binary number:

```
              PHA                    ;space for result
              PushPtr DecString      ;Pointer to decimal string
              PushWord #6            ;Size of string
              PushWord SignedFlag    ;0 = unsigned, 1 = signed
              _Dec2Int
              PopWord BinWord        ;get the result
              RTS

    DecString DC    C' 2342'         ;Number to convert
```

Notice that Dec2Int returns the 16-bit binary result on the stack.

Dec2Long converts the ASCII string for a decimal number to a 32-bit unsigned or signed decimal number:

```
              PHA                    ;space for result
              PHA
              PushPtr DecString      ;Pointer to decimal string
              PushWord #11           ;Size of string
              PushWord SignedFlag    ;0 = unsigned, 1 = signed
```

```
          _Dec2Long
          PopLong BinLong      ;get the result
          RTS

DecString DC    C'-21882423323' ;Number to convert
```

Dec2Long returns the 32-bit result on the stack.

For both Dec2Int and Dec2Long, SignedFlag is a Boolean indicating whether the number is unsigned (0) or signed (nonzero).

APPENDIX 5

ProDOS File Type Codes

The table below indicates the general contents of a file with a given ProDOS file type code. The standard three-character mnemonics are usually shown only when you catalog the disk under ProDOS 8 or APW. Even then, some of the mnemonics, particularly the ones for Apple III SOS files, are not used; instead, the hexadecimal file type code is displayed to identify the file type.

Table of ProDOS File Type Codes

File Type Code	*Standard Mnemonic*	*Description of File*
$00		Uncategorized file
$01	BAD	Bad block file
$02	PCD	Pascal code (SOS)
$03	PTX	Pascal text (SOS)
$04	TXT	ASCII textfile
$05	PDA	Pascal data (SOS)
$06	BIN	General binary file
$07	FNT	Font file (SOS)
$08	FOT	Graphics screen file
$09	BA3	Business BASIC program (SOS)
$0A	DA3	Business BASIC data (SOS)
$0B	WPF	Word processor file (SOS)

File Type Code	*Standard Mnemonic*	*Description of File*
$0C	SOS	SOS system file
$0D–$0E		[Reserved for SOS]
$0F	DIR	Subdirectory file
$10	RPD	Record Processing System data (SOS)
$11	RPI	Record Processing System index (SOS)
$12		AppleFile discard file (SOS)
$13		AppleFile model file (SOS)
$14		AppleFile report format file (SOS)
$15		Screen library file (SOS)
$16–$18		[Reserved for SOS]
$19	ADB	AppleWorks database file
$1A	AWP	AppleWorks word processing file
$1B	ASP	AppleWorks spreadsheet file
$1C–$AF		[Reserved]
$B0	SRC	APW source code
$B1	OBJ	APW object code
$B2	LIB	APW library
$B3	S16	ProDOS 16 system program
$B4	RTL	APW run-time library
$B5	EXE	APW executable shell application
$B6	STR	ProDOS 16 permanent init (start-up) file
$B7	TSF	ProDOS 16 temporary init file
$B8	NDA	New desk accessory
$B9	CDA	Classic desk accessory
$BA	TOL	Tool set
$BB	DRV	ProDOS 16 device driver
$BC–$BE		[Reserved for ProDOS 16 load files]

File Type Code	*Standard Mnemonic*	*Description of File*
$BF	DOC	ProDOS 16 document file
$C0	PNT	Compressed super high-res picture file
$C1	PIC	Super high-res picture file
$C2–$C7		[Reserved]
$C8	FON	ProDOS 16 font file
$C9–$EE		[Reserved]
$EF	PAS	Pascal area on a partitioned disk
$F0	CMD	ProDOS 8 added command file
$F1–$F8		ProDOS 8 user-defined files
$F9	P16	ProDOS 16 file
$FA	INT	Integer BASIC program
$FB	IVR	Integer BASIC variables
$FC	BAS	Applesoft BASIC program
$FD	VAR	Applesoft BASIC variables
$FE	REL	EDASM relocatable code file
$FF	SYS	ProDOS 8 system program

NOTE: SOS stands for the Apple III Sophisticated Operating System.

APPENDIX 6

Memory Cards for the Apple IIGS

A standard GS comes with 256K of RAM, twice the memory capacity of either the IIe or IIc. This configuration is perfectly fine for running IIe/IIc-style programs. It is not enough, however, to run the new GS applications that use the Macintosh-like desktop environment provided by the GS's software tool sets. This is because these types of programs typically need plenty of memory to hold RAM-based tool sets, custom fonts, desk accessories, and so on.

For some applications, you may not need more memory, but the application will perform much better if you do have it. The best example is version 2.0 of AppleWorks. It uses extra memory to increase its desktop space so that you can have several large files open at the same time.

You may also want to add extra memory to take advantage of useful utility programs that make the GS faster and more pleasurable to use. Here are some of the things for which extra memory can be used:

- A RAM disk
- A disk-caching area
- A print-spooling area
- A buffer area for copying disks in fewer passes

RAM disk support for extra memory is a built-in feature on the GS. Using the Control Panel's RAM Disk command, you can allocate as much available memory as you like for use as a RAM disk.

This appendix reviews available memory cards for the GS that plug into the GS's memory expansion slot. See the list of manufacturers below. RAM memory on a card in this slot forms part of the 65816 address space, so programs can use it just as they would use the 256K memory core. The permitted memory limit for such cards is 8 megabytes.

MANUFACTURERS OF MEMORY CARDS FOR THE GS

Apple IIGS Memory Expansion Card
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408/996–1010

GS RAM and GS RAM Plus
Applied Engineering
P.O. Box 798
Carrollton, TX 75006
214/241–6060

RamStakPlus
AST Research Inc.
2121 Alton Avenue
Irvine, CA 92714–4992
714/553–0340

OctoRAM
MDIdeas, Inc.
1163 Triton Drive
Foster City, CA 94404
415/573–0580

RamPak 4GS
Orange Micro, Inc.
1400 N. Lakeview Ave.
Anaheim, CA 92807
714/779–2772

APPLE IIGS MEMORY EXPANSION CARD

The first memory card available for the GS was Apple's own Apple IIGS Memory Expansion Card. It comes with 256K of RAM, but you can expand it to either 512K or 1M by adding one or three additional rows of eight 256Kx1 bit memory chips to empty sockets on the card. (Apple does not recommend a 768K configuration because some programs cannot handle it.) These chips are inexpensive.

The Apple card is definitely a "plain-vanilla" product. It comes with no software, not even a RAM diagnostic program for testing for bad memory chips. In addition, it cannot be expanded past 1M and you cannot add any ROM to it. Despite this, it is an attractive card because of its low price and its excellent owner's guide. The owner's guide clearly illustrates how to install the card and add memory to it, and it contains a useful chapter describing how to allocate memory on the card for use as a RAMdisk.

APPLIED ENGINEERING GS RAM

Applied Engineering is a well-known manufacturer of RAM cards for Apple II computers. Its two main products, RamFactor and (for the IIe only) RamWorks, have sold in the tens of thousands.

Applied Engineering actually has two such cards: GS RAM and GS RAM Plus. The GS RAM uses 256Kx1-bit RAM chips and can hold up to 1.5M of memory. It is essentially the same as Apple's card but with two extra rows of eight RAM sockets.

GS RAM Plus looks very similar to GS RAM, but it uses 1,024Kx1 bit (one-megabit) RAM chips. This means that each row on the card holds one megabyte of memory, so the capacity of the card is six megabytes. A set of eight of these chips is still more expensive than the 256Kx1 chips, but the cost is dropping fast.

Both cards contain an expansion port that Applied Engineering plans to use for adding 2M more RAM, or ROM, via a piggyback card.

GS RAM and GS RAM Plus come with diagnostic software and the AppleWorks 2 Expander program. The Expander enhances AppleWorks 2.0 so that it will work even better than it already does with GS memory cards. The special enhancements it offers are as follows:

- More lines in the word processor (22,600 versus 7250)
- More records in the database (22,600 versus 6350)
- More lines in the word processor and database clipboard (2,042 versus 250)
- Display of the current time on the screen
- Saving large files to multiple disks
- Up to a 64K print buffer (versus 2K)
- Quick time entry in the database

MDIDEAS OCTORAM

The MDIdeas OctoRAM is an interesting memory card. It contains eight standard SIMM (Single In-line Memory Module) sockets that can hold 256K memory modules or 1M memory modules. Thus, the capacity of the card is either 2M (eight 256K SIMMs) or 8M (eight 1M SIMMs).

A SIMM is a small printed circuit card that has eight memory chips soldered to it. To add it to the memory card, you simply clip it into a socket on the card. SIMMs are new to the Apple II world, but they are being used by Apple on the Macintosh Plus. The Macintosh Plus has four SIMM sockets to which you can add 256K SIMMs (1M total) or 1M SIMMs (4M total).

The 1M SIMMs are currently a bit more expensive than conventional memory chips, and they are more difficult to find because they are made by fewer manufacturers. It is expected that they will drop in price, but at a slower rate than the 1,024Kx1 RAM chips.

ORANGE MICRO RAMPAK 4GS

Orange Micro's memory card is called the RamPak 4GS. As its name suggests, its maximum capacity is 4M. It is the only memory card that uses 256Kx4-bit memory chips, which means that you can increase memory by 256K by adding only two chips. This makes the board less crowded than cards that use 256Kx1 RAM chips (eight 256Kx1 chips are needed for 256K). Two 256Kx4 chips are still more expensive than eight 256Kx1 chips, however, because 256Kx4 chips are a nonstandard size. They are also more difficult to find.

RamPak comes with a Memory Management Utilities disk that contains a useful and easy-to-use program that performs the following tasks:

- Memory verification
- Reporting of disk-access statistics
- Disk caching

This program installs itself as a Classic Desk Accessory (such as the Control Panel), so you can call it up at any time by pressing Control-OpenApple-Esc from the keyboard.

The most interesting aspect of the RamPak software is its ability to enable disk caching for any disk device used with ProDOS. Caching is the storing of disk blocks in memory so that when a read operation occurs, the block can be retrieved quickly from memory rather than from the relatively slow, mechanical drive. If most of the disk is cached, file operations are sped up dramatically.

The main advantage of caching over using a RAM disk is that there is no danger of losing data because only read operations are cached. Data is always transferred to the disk when a write occurs. Another advantage is that you do not have to transfer files to the cache explicitly; that is done transparently by the caching software the first time you load a program into memory.

The RamPak software is able to use up to 3.5M as a cache buffer. You can either tell the software how much memory to use as a cache or tell it to allocate memory dynamically.

AST RESEARCH RAMSTAKPLUS

The AST RamStakPlus is interesting because it is the only memory card that contains sockets for ROM. Its maximum RAM capacity is 1M (four rows of eight 256Kx1 bit chips), the same as Apple's card. It is too bad that it was not designed to hold more RAM, but it would be difficult to do because the ROM sockets take up a lot of board space.

The RamStakPlus has four ROM sockets, each of which can hold a 8K-64K EPROM (Erasable Programmable ROM) or a 2K-32K EEPROM (Electrically Erasable Programmable ROM). You will probably want to use EEPROM because you can erase it and program it simply by running a program. You need a special programming device to program an EPROM.

The RamStakPlus comes with a utility program that performs diagnostics and programs EEPROMs. Programming an EEPROM is simple; all you have to do is put all the files you want to place in the EEPROM in a subdirectory, then run the EEPROM programming program that AST supplies.

In keeping with Apple's standard guidelines. the files in ROM are organized in the same way they are on a disk, so you will have created a ROM disk. If the ROM contains the necessary operating system programs, you can even boot from ROM by setting the Startup Slot in the Control Panel to ROM Disk. The big difference between a ROM disk and a RAM disk is, of course, that the ROM disk does not disappear when you turn off the GS.

MAKING A CHOICE

With this analysis of GS memory cards by your side, you should be able to choose a card that will suit your present and future memory needs. To summarize, the key factors in deciding what card to buy are:

- The base price of the card
- The cost of adding more memory
- The ease of adding more memory
- The upper RAM limit
- The ability to deal with ROM
- The availability of software utilities

Table A6 summarizes the features of each memory card.

If you have a limited budget, keep in mind that you can start with just a little RAM and add more when prices drop (or when you desperately need more). (You can buy RAM chips at electronics parts stores and mail-order warehouses.) At this stage, you will have to determine what types of RAM chips to buy and where to install them on the card.

RAM chips are produced by various manufacturers to many different specifications, so you must be careful to choose the right ones for your card. For GS cards, you need RAM with a speed specification of 150 nanoseconds or faster that uses the "CAS before RAS" refreshing technique. The capacity and configuration of the RAM chips is also important; the possible choices are:

Table A6: Apple IIGS Memory Cards: A Comparison Chart

Product	*Memory Range*	*Minimum Increment*	*RAM Chip Size*	*ROM Capability*	*Utility Software*
IIGS Memory Expansion Card	256K–1M	256K	256Kx1	None	None
GS RAM	256K–1.5M	256K	256Kx1	Option	Diagnostics
GS RAM Plus	1M–6M	1M	1,024Kx1	Option	AppleWorks Enhancements
RamStak Plus	256K–1M	256K	256Kx1	On card	Diagnostics ROM programmer
OctoRAM	256K–2M 1M–8M	256K 1M	256K SIMM 1M SIMM	Option	Diagnostics
RamPak 4GS	512K–4M	256K	256Kx4	None	Diagnostics Caching Statistics

- 256Kx1-bit (8 chips = 256K)
- 1,024Kx1-bit (8 chips = 1M)
- 256Kx4-bit (2 chips = 256K)
- 256K SIMM
- 1M SIMM

Consult your card's manual to determine what kind of RAM chip the card requires and how many chips you need (eight for 256Kx1 or 1,024Kx1; two for 256Kx4; or one SIMM). If you want to play it safe, order the RAM chips directly from the card's manufacturer. You will probably pay a little more, but you will not have any headaches.

The order in which you fill rows of empty RAM sockets on a card is also important. If you do not fill the sockets in the proper order, the card will not work properly. Again, consult your card's manual for installation instructions.

APPENDIX 7

Disk Drives for the Apple IIGS

There are nine different Apple-brand, removable-media disk drives you can use with the Apple IIGS: five 5¼-inch drives and four 3½-inch drives. This appendix looks at each of them and examines how to use them with the GS.

5¼-INCH DISK DRIVES

The original drive for the Apple II was the Disk II, first released in 1979. It works with 140K 5¼-inch floppy disks and comes with a permanent cable that terminates in a 20-pin rectangular connector.

Apple no longer manufactures the Disk II, but the company has four other 5¼-inch drives to replace it: the UniDisk 5.25, the Apple 5.25 Drive, the Disk IIc, and the DuoDisk. (The Apple 5.25 Drive is the same as the UniDisk 5.25 except for its GS-like platinum color.) The drive mechanism in each of these drives is functionally equivalent to that of the Disk II, so all of Apple's 5¼-inch drives read and write the same disks.

The DuoDisk is unique in that it contains two drives in one box. The UniDisk 5.25, Apple 5.25 Drive, and Disk IIc are all one-drive devices. Each of these drives has the same type of connector cable, however—one that terminates in a D-shaped, 19-pin, male DB-19 connector.

You can connect two (but no more than two) UniDisk 5.25 or Apple 5.25 drives together by attaching the second drive to a connector on the back of the first drive. This method of connecting multiple drives is called *daisy-chaining* and is Apple's "official" technique for attaching multiple drives to a system. Because a DuoDisk contains two drives, it does not have a daisy-chain connector; nor do the older Disk II or Disk IIc drives.

3½-INCH DISK DRIVES

In 1985, Apple started selling a 3½-inch drive for the Apple II called the UniDisk 3.5. This is a very intelligent device, containing a 65C02 microprocessor and a program in ROM for controlling disk data transfers. It uses double-sided 800K hard-shelled disks for media and works with any Apple II model (except older Apple IIc systems). The UniDisk 3.5 cable has the same DB-19 connector as the cable for the UniDisk 5.25 and the DuoDisk.

When the GS was announced in September 1986, another 3½-inch drive was unveiled, the Apple 3.5 Drive. It works with the same 800K disks as the UniDisk 3.5 but does not have the built-in intelligence of the UniDisk 3.5. It works only with the GS or the Macintosh. The main advantage of using an Apple 3.5 Drive instead of a UniDisk 3.5 on the GS is that the Apple 3.5 Drive operates slightly faster.

The other two Apple-brand 3½-inch drives that can be made to work with the GS are the standard 400K and 800K external drives for the Macintosh. To use them, you must have a Universal Disk Controller (UDC) from Central Point Software (9700 S.W. Capitol Hwy., #100, Portland, OR, 97219 [503/244–5782]); more information on the UDC is provided below. The UDC also works with all other Apple drives (except the DuoDisk) and with one non-Apple-brand 3½-inch drive, the 800K Chinon 3.5 drive sold by Central Point Software.

USING DRIVES WITH THE APPLE IIGS

The GS has slots like those on the IIe and built-in peripheral ports like those on the IIc. To select whether a port or its corresponding slot is active, use the Slots command in the Control Panel desk accessory. You can call up the desk accessory menu at any time by pressing Control-OpenApple-ESC.

One of the built-in ports, corresponding to slot 5, is a disk drive port called SmartPort; it is designed to handle 3½-inch disk drives and any intelligent peripherals that use the data-exchange protocol and electrical specifications defined by SmartPort. The SmartPort connector is a DB–19 type and is located at the back of the GS.

Another built-in port, corresponding to slot 6, is the 5¼-inch Drive Port. It shares the same physical connector as SmartPort—this does not cause problems because 5¼-inch drives can be daisy-chained to SmartPort devices.

You will probably want to daisy-chain drives to the SmartPort instead of using plug-in controller cards. Five types of drives can be chained to the SmartPort: the UniDisk 3.5, the UniDisk 5.25, the Apple 5.25 Drive, the DuoDisk, and the Apple 3.5 Drive. (You can also use the Disk II, but only if you have a special adapter cable made up.) It is important to understand, however, that you cannot chain these drives in just any order.

If you are using Apple 3.5 Drives, they must appear at the beginning of the chain, before any UniDisk 3.5 drives or 5¼-inch drives. You can use up to two of these types of drives.

At the end of the chain come the 5¼-inch drives: up to two UniDisk 5.25 drives or one DuoDisk (which is equivalent to two UniDisk 5.25 drives) may be used. There is one caveat relating to the DuoDisk, however—because of a hardware problem in the drive, you cannot daisy-chain a DuoDisk having a serial number below 433754. Apple has published a technical note describing a hardware fix to solve this problem, so track it down if you want to use an old DuoDisk with the GS.

Between the Apple 3.5 Drives at the beginning of the chain and the UniDisk 5.25 drives at the end of the chain come any UniDisk 3.5 drives. The number of UniDisk 3.5 drives is limited by the capacity of the GS power supply. Apple recommends that you have no more than four drives connected to the SmartPort (including any Apple 3.5 Drives and 5¼-inch drives), although the GS seems to be able to handle six quite well.

Keep in mind that no other ordering of drives will work: the order must be Apple 3.5 Drives followed by UniDisk 3.5 drives followed by 5¼-inch drives.

The ProDOS 8 operating system requires that disk devices be mapped to traditional slot/drive combinations with no more than two drives per slot. How, then, does it deal with the possibility that more than two drives may be chained to the slot 5 SmartPort?

The way ProDOS 8 assigns slot/drive combinations to SmartPort disk drives is a bit unusual. In most situations, the first four 3½-inch drives are considered to be the slot 5/drive 1, slot 5/drive 2, slot 2/drive 1, and slot 2/drive 2 devices. The two 5¼-inch drives on the SmartPort correspond to slot 6/drive 1 and slot 6/drive 2. That is, connecting them to the end of the SmartPort chain is the same as connecting them to a controller card in slot 6.

If you use the GS Control Panel to define a system RAM disk, however, the slot/drive assignments are scrambled slightly. The first 3½-inch drive is still the slot 5/drive 1 device, but the RAM disk is assigned as the slot 5/drive 2 device. The next two 3½-inch drives "occupy" slot 2/drive 1 and slot 2/drive 2. Any other 3½-inch drives are not recognized by ProDOS 8.

ProDOS 16, on the other hand, does not require slot/drive assignments and will work with any number of disk devices without difficulty.

If you wish, you can also use either of the two standard Apple 5¼-inch disk controller cards on the GS. If you put the controller card in slot 5 or 6, keep in mind that you will not be able to use the SmartPort or the Drive Port, respectively. You can also use Central Point Software's Universal Disk Controller on the GS. It works with almost any type of 5¼-inch and 3½-inch drive available for the GS.

The Chinon 3.5 drives the UDC works that are also sold by Central Point Software. The only major differences between them and Apple's 3½-inch drives are that they do not have daisy-chain connectors and that the disk eject button is completely mechanical. (There is an electronic disk eject mechanism that is controlled by the same software commands as the Apple 3½-inch drives.) The Chinon drives also work nicely as 800K external drives for the Macintosh.

A nice feature of the GS is that you can use the Control Panel to specify the slot of the drive from which to boot—this is called the Startup Slot. The default is "Scan," which means that the GS will boot from the first drive it encounters in a search beginning in slot 7 and moving down to slot 1 (this is the method used on the IIe). Only the first drive in each slot or port is considered, however, so you cannot boot from a secondary drive. If you prefer, you can tell the GS to boot from a given slot—specify slot 5 to boot from the first 3½-inch drive connected to the SmartPort, for example.

APPENDIX 8

Resource Material

If you are interested in developing software on the GS you should become a member of APDA, the Apple Programmer's and Developer's Association. Its address and telephone number are as follows:

Apple Programmer's and Developer's Association
290 S.W. 43rd Street
Renton, WA 98055
206/251–6548

APDA is an association that was created by Apple Computer, Inc. in August 1986. It is administered by the Apple Puget Sound Program Library Exchange Co-operative Association (A.P.P.L.E. Co-op). The primary function of APDA is to disseminate prerelease versions of official Apple programming information for the Apple II and Macintosh families of computers. This includes technical reference manuals, technical notes, and software development tools. APDA is also a convenient source of non-Apple software and books for software developers.

The advantage of joining APDA is that you can get your hands on useful technical information quickly. You do not have to wait for the information to be polished and published in final form.

DEVELOPMENT SOFTWARE

To develop software on the GS, you will probably want the Apple IIGS Programmer's Workshop (APW), which is available from APDA (see above). The basic APW package includes an editor, 65816 assembler, linker, and several support programs you can use to create programs that run under ProDOS 16. The official APW development system is almost identical to the ORCA/M system available from The Byte Works Inc., #207–4700 Irving Blvd. NW, Albuquerque, NM, 87114 (505/898–8183). This is because The Byte Works developed APW for Apple.

Another alternative GS assembler, which does not require APW, is Merlin 816 from Roger Wagner Publishing, Inc. (1050 Pioneer Way, Suite P, El Cajon, CA, 92020 [619/442–0522]). It comes with a linker which is capable of creating standard ProDOS 16 applications.

Apple also distributes an APW-compatible C compiler (developed by Megamax, Inc.) which generates object code in the format expected by the APW linker.

An APW-compatible Pascal compiler is available from TML Systems (#23–4241 Baymeadows Road Jacksonville, FL, 32217 [904/636–8592]). TML also sells a stand-alone Pascal compiler that uses the standard desktop environment of the GS.

MACINTOSH CROSS-COMPILERS

You can also develop GS applications using cross-compilers on the Macintosh. The advantage of doing this is primarily speed: compilation times are many times faster when performed by the Macintosh's 68000 microprocessor than when performed by the GS's less powerful 65816.

Consulair Corp. (140 Campo Drive, Portola Valley, CA, 94025 [415/851–3272]) sells the MACtoGS Assembler/Linker system. It assembles and links APW-compatible 65816 assembly language source code files on a Macintosh to create complete GS applications. Of course, to run the applications, you must first transfer them to the GS (using a communications program or Apple's Passport program).

TML Systems (see above) has a Pascal cross-compiler that generates 65816 source code files that can then be processed by Consulair's MACtoGS Assembler/Linker. You can buy the compiler from TML separately or with the MACtoGS Assembler/Linker included. Another Pascal cross-compiler is available from Megamax, Inc. (Box 851521, Richardson, TX, 75085 [214/987–4931]).

Megamax also has a C cross-compiler which works with the same source code as the version of C it developed for APW does.

REFERENCE BOOKS

The many books you will find useful for teaching yourself how to program the GS fall into four categories:

- GS-specific books
  At the present time, most of these books have been written by Apple Computer, Inc. Those that are not yet in final form are available from APDA. When completed, most will be published and distributed by Addison-Wesley Publishing Company, Inc.

- Books about the IIe and IIc
  These books will assist you in developing traditional applications on the GS.

- General books on programming in 65816 assembly language
- Books about the Macintosh
  These books are useful because they describe general techniques for programming in a desktop environment.

Books for the Apple IIGS

Apple Computer, Inc., *Apple IIGS Firmware Reference* (Reading, MA: Addison-Wesley Publishing Company, 1987).

Apple Computer, Inc., *Apple IIGS Hardware Reference* (Reading, MA: Addison-Wesley Publishing Company, 1987).

Apple Computer, Inc., *Apple IIGS ProDOS 16 Reference* (Reading, MA: Addison-Wesley Publishing Company, Inc., 1987).

Apple Computer, Inc., *Apple IIGS Toolbox Reference*, volumes I and II (Reading, MA: Addison-Wesley Publishing Company, 1988).

Apple Computer, Inc., *Apple Programmer's Workshop Assembler Reference* (Cupertino, CA: Apple Computer, 1987).

Apple Computer, Inc., *Apple Programmer's Workshop Reference* (Cupertino, CA: Apple Computer, 1987).

Apple Computer, Inc., *Human Interface Guidelines* (Reading, MA: Addison-Wesley, 1987).

Apple Computer, Inc., *Programmer's Introduction to the Apple IIGS* (Reading, MA: Addison-Wesley Publishing Company, 1988).

Apple Computer, Inc., *Technical Introduction to the Apple IIGS* (Reading, MA: Addison-Wesley Publishing Company, 1986).

Fischer, Michael, *The Apple IIGS Technical Reference* (Berkeley, CA: Osborne McGraw-Hill, 1987).

Wagner, Roger, *Apple IIGS Machine Language for Beginners*, (Greensboro, NC: Compute! Publications, 1987).

Books for the Apple IIe and IIc

Apple Computer, Inc., *Apple IIc Technical Reference Manual* (Reading, MA: Addison-Wesley Publishing Company, 1985).

Apple Computer, Inc., *Apple IIe Technical Reference Manual* (Reading, MA: Addison-Wesley Publishing Company, 1985).

Apple Computer, Inc., *Apple Numerics Manual* (Reading, MA: Addison-Wesley Publishing Company, 1986).

Little, Gary B., *Apple ProDOS: Advanced Features for Programmers* (Bowie, MD: Brady Communications Company, 1985).

Apple Computer, Inc., *ProDOS Technical Reference Manual* (Reading, MA: Addison-Wesley Publishing Company, 1985).

Little, Gary B., *Inside the Apple IIc* (Bowie, MD: Brady Communications Company, 1985).

Little, Gary B., *Inside the Apple IIe* (Bowie, MD: Brady Communications Company, 1985).

Books on 65816 Assembly Language Programming

Eyes, David, and Ron Lichty, *Programming the 65816 Including the 6502, 65C02 and 65802* (New York: Prentice-Hall, 1986).

Fischer, Michael, *65816/65802 Assembly Language Programming* (Berkeley, CA: Osborne McGraw-Hill, 1986).

Books for the Macintosh

Apple Computer, Inc., *Inside Macintosh*, volumes I, II, III, and IV (Reading, MA: Addison-Wesley Publishing Company, 1985, 1986).

Chernicoff, Stephen, *Macintosh Revealed: Unlocking the Toolbox* (Indianapolis, IN: Hayden Book Company, 1985).

Chernicoff, Stephen, *Macintosh Revealed: Programming with the Toolbox* (Indianapolis, IN: Hayden Book Company, 1985).

Little, Gary B., *Mac Assembly Language: A Guide for Programmers* (New York: Prentice-Hall, 1986).

MAGAZINES

Magazines are the best source of information on what software and hardware products are currently available for the GS. Many of the magazines that are available are written for a general audience, but some are clearly for programmers only. The ones marked with an asterisk in the following list are the ones programmers will find most interesting.

A+
11 Davis Drive, Belmont, CA 94002
415/598–2290

**Apple Assembly Line*
P.O. Box 280300, Dallas, TX 75228
214/324–2050

Call -A.P.P.L.E.
290 S.W. 43rd St., Renton, WA 98055
206/251–5222

inCider: The Apple II Magazine
80 Elm Street, Peterborough, NH 03458
603/924–9471

**MacTutor: The Macintosh Programming Journal*
P.O. Box 400, Placentia, CA 92670
714/630–3730

**Nibble: The Magazine for Apple II Enthusiasts*
45 Winthrop Street, Concord, MA 01742
617/371–1660

**Open Apple*
P.O. Box 7651, Overland Park, KS 66207

INDEX

Program Disk for

Exploring the Apple IIGS

by Gary B. Little

All of the programs listed in this book are available on disk, in source code form, directly from the author. The disk also contains several other programs, including some useful desk accessories and a program to display and print screen image files.

TO ORDER the disk, simply clip or photocopy this entire page and complete the coupon below. Enclose a check or money order for $20.00 in U.S. funds. (California residents add applicable state sales tax.)

MAIL TO:

GARY B. LITTLE
1925 Bayview Avenue
Belmont, CA 94002

Please send me:

_______ (quantity) Disks to accompany ***Exploring the Apple IIGS, by Gary B. Little,*** at $20.00 each.

_______ Check enclosed.

YOUR SIGNATURE: ______________________________________

Name: ________________________ Title: ____________________

Company (if applicable): ______________________________

Address: ____

City: ________________ State: ____________ Zip: _____________

>$22.95 FPT USA

GARY B. LITTLE

Addison-Wesley Publishing Company

EXPLORING THE APPLE IIGS™

Best-selling author Gary B. Little reveals the power of the amazing new Apple IIGS

The Apple IIGS™ is the sophisticated new member of the Apple® II family of computers with outstanding color graphics and sound capabilities. It features a powerful 16-bit, 65816 microprocessor, custom sound and graphics chips, an enhanced operating system called ProDOS® 16, and a Macintosh™-like Toolbox of programming tools to help programmers easily create windows, pull-down menus, dialog boxes, and other user interface components.

Now Gary B. Little, the author of the acclaimed **Inside the Apple IIe** and **Inside the Apple IIc** has written an in-depth, technical introduction to the inner workings of the Apple IIGS. Written for all assembly language and Applesoft BASIC programmers, **Exploring the Apple IIGS** is a detailed study of how the Apple IIGS is built and how it works.

Little provides thorough discussions of:

- the architecture and capabilities of the 65816 microprocessor
- software development environments and utilities
- the Apple IIGS Programmer's Workshop
- file management with ProDOS 16
- memory map and memory management
- using Super Hi-Res graphics
- event handling
- using the Apple IIGS Toolbox

Little's easy-to-read style makes the intricacies of programming the Apple IIGS readily accessible. He analyzes all the major functions a programmer will have to know and shows how to assemble them into a complete application. Each chapter features extensive examples of program code. **Exploring the Apple IIGS** is the ideal guide to learning to program the exciting Apple IIGS.

Gary B. Little is a well-known author, columnist, and software developer. He is a leading authority on the Apple II family of computers. His books include the best-selling **Inside the Apple IIe, Inside the Apple IIc,** and **Apple ProDOS: Advanced Features for Programmers.** Gary is the developer of the Point-to-Point communications program and he is also a Contributing Editor for *A + Magazine*. He practices computer law in Vancouver, British Columbia.

Cover design by Doliber Skeffington

ISBN 0-201-15539-7